

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

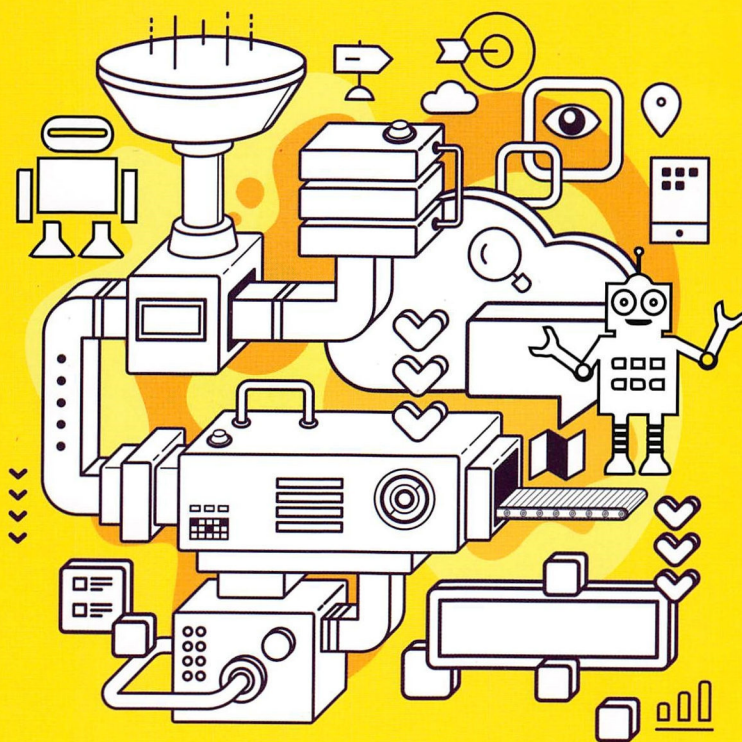
智能运维即未来 AIOps时代开启
腾讯|滴滴|美团|微博|清华等产学研界翘楚瞩目力荐

Broadview[®]
www.broadview.com.cn

智能运维

从0搭建大规模分布式AIOps系统

彭冬 朱伟 刘俊 等著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

► 本书作者

彭冬：微博广告基础架构团队负责人、技术专家，商业大数据平台及智能监控平台发起人，目前负责广告核心引擎基础架构、Hubble 智能监控系统、商业基础数据平台（D+）等基础设施建设。关注计算广告、大数据、人工智能、高可用系统架构设计、区块链等方向。在加入微博之前，曾就职于百度负责大数据平台建设，曾担任趣点科技联合创始人兼 CTO 等职位。毕业于西北工业大学，曾在国内外知名期刊发表多篇学术论文，拥有 9 项发明专利。

朱伟 @kimi：微博广告 SRE 团队技术负责人，高级运维工程师，2016 年 4 月加入微博，目前主要负责微博广告智能监控报警平台和服务治理等项目的建设与研究。

刘俊：微博平台部监控技术负责人，负责微博平台、PC 微博大规模监控系统的建设，主要关注实时大数据、运维自动化、智能化方向。2014 年加入微博，之前曾在新浪、搜狐等公司从事运维监控方面的工作。

王莉：University of Georgia 硕士研究生，主要研究用机器学习方法，识别植物被水淹没的季节性规律，研究成果已发表在 SCI 高影响因子期刊。2017 年加入微博广告团队，致力于用数据分析和机器学习模型，优化广告业务策略，洞悉商业价值。

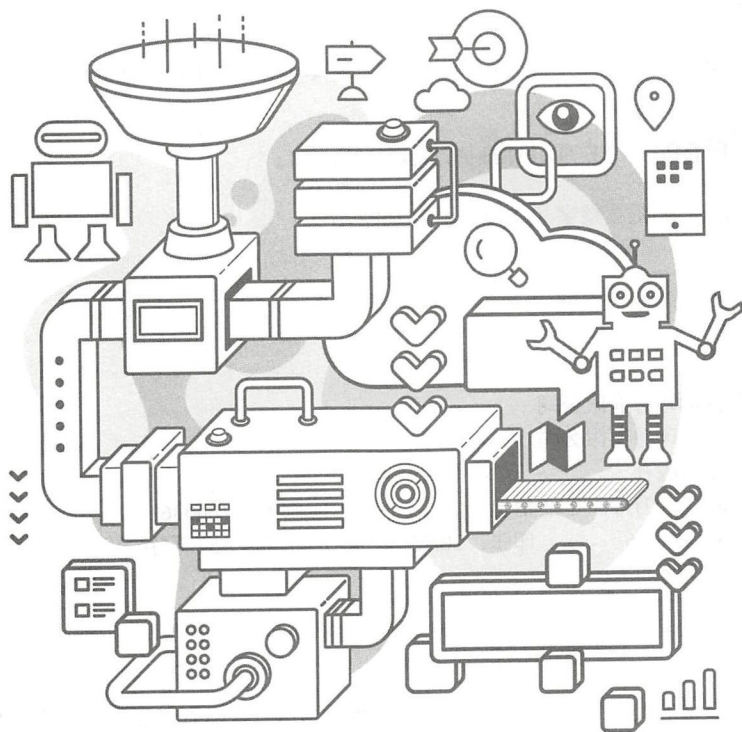
陆松林：微博广告数据仓库负责人，高级研发工程师，先后就职于搜狐、爱奇艺，主要研究数据仓库、数据治理相关技术。

车亚强：微博广告大数据开发工程师，曾在百度外卖负责实时流、微服务相关研发工作，目前主要研究方向为实时流、微服务架构设计。

智能运维

从0搭建大规模分布式AIOps系统

彭冬 朱伟 刘俊 王莉 陆松林 车亚强 著



电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书结合大企业的智能运维实践，全面完整地介绍智能运维的技术体系，让读者更加了解运维技术的现状和发展。同时，帮助运维工程师在一定程度上了解机器学习的常见算法模型，以及如何将它们应用到运维工作中。

全书共分4篇。第1篇运维发展史，重点阐述当前运维的发展现状及面临的技术挑战；第2篇智能运维基础设施，重点讲述大数据场景下的数据存储、大数据处理和分析的方法与经验，以及海量数据多维度多指标的处理分析技术；第3篇智能运维技术详解，重点关注在新时期大数据时代下智能化的运维技术，包括数据聚合与关联、数据异常点检测、故障诊断和分析、趋势预测算法；第4篇技术案例详解，为大家梳理了通过开源框架 ELK 快速构建智能监控系统的整体方案，还将分享微博平台和微博广告两个不同业务场景下智能监控系统的技术实践。

本书适合运维、开发、架构、DevOps 工程师及广大互联网技术爱好者研读和借鉴。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

智能运维：从0搭建大规模分布式 AIOps 系统 / 彭冬等著. —北京：电子工业出版社，2018.7
ISBN 978-7-121-34663-7

I. ①智… II. ①彭… III. ①软件维护 IV. ①TP311.53

中国版本图书馆 CIP 数据核字（2018）第 141688 号

策划编辑：张春雨

责任编辑：葛 娜

印 刷：三河市良远印务有限公司

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16

印张：20.75

字数：457 千字

版 次：2018 年 7 月第 1 版

印 次：2018 年 7 月第 1 次印刷

印 数：4000 册

定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zits@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

推荐语

本书和国内外第一个《企业级 AIOps 实践建议白皮书》在很多内容上不谋而合，也是对 AIOps 白皮书的深度细化和技术补充。本书基于新浪微博平台技术部门及广告技术部门的生产实践，有理有据，有说服力。文字简练、表达清晰。相关实践可落地，让有志于实践 AIOps 的企业及运维同仁“开箱即用”。

——**萧田国** 高效运维社区发起人、AIOps 标准及白皮书发起人

伴随着大数据和机器学习技术的发展，智能运维（AIOps）也成为运维领域极大的热点。本书不仅介绍了智能运维的发展过程，还针对大数据处理和智能运维相关的底层技术进行了详细的分析，并且结合微博的具体应用场景提供了在实时监控、报警、异常检测、故障根源分析、趋势预测、数据关联等方面的实战案例，浓缩了大量开发知识和实践经验。这是一本非常有参考价值的智能运维著作。

——**裴丹** 清华大学计算机系长聘副教授、青年千人、
美国 AT&T 研究院前主任研究员、智能运维算法专家

当下，世界已经在全面向智能化迈进。可以说，所有既有的“传统”工作，都可以用智能化思路来改造。从反向的角度讲，如果不能智能化工作，将面临逐渐被淘汰的局面。当然，作为支撑整个 IT 世界地基的运维工作，也在其中。本书的作者团队在运维体系、系统架构、大数据及 AI 等方面具有多年的非常丰富的实战经验，他们对智能运维技术体系进行了全面的梳理，将从思路到工具再到实践的全过程在本书中完整呈现。做面向未来的智能运维，做不被机器淘汰的运维人。在此特向读者推荐本书。

——**王鹏云** 多盟联合创始人，蓝色光标技术创新孵化中心总经理，
曾任 139 移动互联研发总监、魔时网 CTO

IV 智能运维：从 0 搭建大规模分布式 AIOps 系统

伴随着云计算和 IoT 技术在各行各业的普及与发展，企业的运维规模不断扩大，我们可以十分确定地预见到，从自动化运维演进到智能化运维，是运维技术发展的必然趋势。本书是从运维大数据技术到 AI 运维技术的书籍，详细介绍了实施智能运维依赖的基础设施和架构技术，既包括通用的智能运维技术，又通过实战案例引出具体的技术在运维领域的应用与效果，兼具参考性与实操性，是大家升级运维能力过程中不可错过的一本好书。如果你对运维领域感兴趣，本书将是你的不二之选。

——梁定安 腾讯运维技术总监、专家工程师

当大数据、人工智能等技术赋能传统行业、驱动变革时，运维+AI 则把 IT 行业中原来完全依靠人工经验、重复枯燥的工作，推向了自动化的智能运维，引领运维的下一步发展方向。智能运维极大减轻了运维工程师的负担，但也对他们提出了新的技术能力要求。本书主要从发展历史、技术体系以及实际应用等方面梳理了智能运维体系，使读者能够理解与掌握智能运维的知识与方法。本书可以作为运维工程师提升运维水平的重要参考，也可以作为运维工程师职业发展的参考。

——钟华 美团打车技术研发部负责人

智能运维已经成为运维领域公认的未来，所有人都想知道自己怎么做才能赶上智能运维的浪潮。本书介绍了作者在异常检测、根因分析、时序预测等智能运维领域的实践经验，更重要的是，书中整理和讲解了智能运维的两大基石：大数据和机器学习。相信读者阅读此书必有所获。

——饶琛琳 日志易产品总监，前新浪微博系统架构师

终于看到彭冬写的这本书！从运维平台的大数据处理到架构设计原理，再到 AIOps 的相关模型和算法，并将智能运维工程架构与算法实践相结合，本书非常具有参考价值。

——陈晓峰 火币集团副总裁

随着大数据时代的到来，在海量数据场景下，AIOps 开始成为下一代运维技术发展的热点。本书系统地介绍了大数据采集、存储、处理、计算以及策略应用的各个环节，并以微博监控为实际案例向读者展示了监控平台建设的实践经验。相信有志于 AIOps 方向的读者，读过此书后都能有自己的体会和收获。

——陆沛 滴滴打车 SRE 团队负责人、技术专家

推荐序 1：运维的时代化变迁

互联网刚兴起的时候，运维还只是一个简单的服务安装管理及监控工作，没人会想到人类在互联网上建立了如此庞大的业务生态。从衣食住行到教育金融，服务器的规模在急剧膨胀，从简单的人力可管控，逐渐进化到依赖自动化体系来管理，但是另一方面，仅依赖工具已经不能很好地解决运维场景的需求。

根据正式的定义，智能运维通常用 AIOps (Algorithmic IT Operations) 这个专有名词来表达，是指利用大数据分析、机器学习等人工智能技术来自动化管理运维事务。早期的大规模运维以服务管理及监控为目标，自动化运维工具可以满足绝大部分需求，大数据采集及分析主要应用在服务监控及分析方面。但随着服务规模的膨胀，通过人工来管理大规模服务已经力不从心，也很难达到服务可用性的要求。2017 年，Gartner 也发布了 AIOps 的市场定义，并预测到 2019 年，全球 25% 的企业将用 AIOps 系统替代传统运维管理系统。在 2017 年这个比例低于 5%，而到 2022 年将超过 40%。

微博作为国内典型的互联网服务之一，在智能运维的实践方面获得了很多经验。比如每年的春晚，有大量用户在微博上进行祝福与互动，微博的 Feed 项目、广告系统、搜索等服务的自动化扩缩容，通过数据标准化算法分析出 QPS 和慢速比，并根据实时压测反馈的数据生成水位线，结合两个指标和水位线的波动情况进行自动扩容和缩容。

在日常的运维工作中，智能运维也在多方面发挥重要作用，比如告警收敛，利用智能算法过滤，收敛大量无效、重复的告警信息，通过聚类算法将大量、多维度的告警聚合为少量事件，通过告警分类算法提高准确率和减少误报。另外，智能运维还被大量应用于故障定位及服务自动修复方面，基于服务运行的日志及告警数据，实现非人工干预的自动化处理，比如自动摘除、重启等操作。

AIOps 已经成为运维领域的发展趋势，但是目前可供参考的书籍较少，且大多偏理论及小规模服务，而针对真正大规模线上实操的书籍非常匮乏。微博技术团队的彭冬、刘俊、朱伟等同事，长期从事微博运维数据相关技术研发，对于如何将 AI 理论和技术应用于大规模服务管理

VI 智能运维：从 0 搭建大规模分布式 AIOps 系统

方面颇有心得。在本书中，系统地介绍了大数据运维基础理论知识，如数据采集、分布式消息队列、实时计算框架、时序数据库等，并结合大数据人工智能技术如 TensorFlow 机器学习框架、趋势预测算法等，介绍了微博平台、微博广告在 AIOps 上的具体实操。相信用心阅读的读者，可以从中深入了解到他们在这领域的领悟。

目前 AIOps 在业界也刚刚开始，微博技术团队也乐于和广大开发人员分享微博在 AIOps 实践中的心得，欢迎大家关注@微博平台架构和@微博技术学院，了解后续相关的公开技术活动。

杨卫华 (Tim Yang)
微博研发副总经理

推荐序 2：运维的新视角

最近 20 年，互联网特别是移动互联网的普及让用户的获取突破了地理的限制，各个领域都出现了亿级乃至十亿级海量用户规模的高科技公司，如搜索领域的 Google、百度，社交领域的 Facebook、腾讯，出行领域的 Uber、滴滴，电商领域的 Amazon、阿里巴巴，以及中国最大的自媒体社交平台微博等。

服务海量用户需要超大规模的在线分布式系统和大数据处理系统。同时，在竞争日益激烈的市场环境下，高科技公司产品和技术快速迭代能力也至关重要。

在国外，运维团队是 infrastructure 很重要的组成部分，比如 Google 的 SRE（Site Reliability Engineer）团队，就是从运维的角度为整个公司产品可用性服务的。在国内，近几年创业氛围非常好，2014 年出现了大量的 O2O 创业公司，2015 年直播和短视频企业大量崛起，2016 年共享单车等共享经济方向的创业公司非常红火，2017 年区块链几乎占领了大部分创业板块。对于这些创业企业而言，产品的快速迭代、极致的用户体验都是至关重要的，而在这个过程中运维团队就非常重要。

对于微博这样的成熟型企业，运维体系的完备性和先进性是必不可少的。从运维平台化到智能化，能让产品迭代更加高效，同时保障系统的稳定。微博作为拥有全球超过 4 亿月活跃用户的平台，其自身具有非常显著的特点：媒体属性和社交特性。微博传承了新浪的媒体属性，在媒体和新闻传播上有着非常重要的价值，而社交特性让这种信息的传播产生爆炸式的效应，可以看到，热门事件和重要的新闻资讯几乎第一时间都是在微博上进行传播的。同时，社交属性也体现在明星和大 V 效应上，一些行业的意见领袖拥有百万甚至千万级别的粉丝量，每天在微博上进行转发、评论等互动的行为非常频繁。正由于微博的这些特点，微博广告作为寄生于微博平台上的商业产品，也继承了相同的属性，同样也拥有相当复杂的业务形态。正是在这种复杂的业务场景下，大规模的传播效应、海量的数据、极其复杂的社交关系、多变的热门事件对整个广告系统的稳定性是一个极大的考验。那么如何在保证产品技术快速迭代能力的同时，保证系统的高可靠性和高可用性呢？彭冬领导的微博广告平台运维团队给出的答案是 AIOps。

VIII 智能运维：从 0 搭建大规模分布式 AIOps 系统

通过 Big Data、AI/ML 等技术，AIOps 能更快更好地收集和分析超大规模系统的运行状态，并能大幅提升运维系统的自动化和智能化水平，从而提升运维工作的整体效率。

彭冬领导的团队搭建了一套这样的 AIOps 系统，并取得了明显的效果。这套智能运维系统在微博广告平台产品技术快速迭代的同时，提升了系统的可靠性和可用性，有力地支撑了微博广告收入的快速增长，智能运维技术在微博复杂的场景下得到了很好的实践。

很高兴彭冬及其团队成员能抽出时间编写这本书，将他们在智能运维上的架构设计思路和实践经验分享给更多的互联网界朋友。这本书是他们多年积累的工作经验的总结，书中有大量案例和前瞻性研究，无论是工程架构还是模型算法，对读者都具有借鉴价值，也希望这本书能成为一座桥梁，促进更多的技术交流，从而让大家共同成长。

张志强

微博广告业务部总经理

推荐序 3：大数据时代的运维

在互联网时代，尤其是社交网络、电子商务与移动通信把人类社会带入一个 PB 级别以上单位的结构与非结构信息的大数据时代。数据量的爆发性增长，使企业 IT 架构不断扩展，服务器、存储设备的数量越来越多，网络也变得更加复杂。而大数据的 4V 特征，数据量大 (Volume)、类型繁多 (Variety)、价值密度低 (Value)、时效高 (Velocity) 也使得传统的技术架构和路线，已经难以高效地处理如此海量的数据。可以说，大数据时代对企业的数数据驾驭能力提出了新的挑战。尤其是大数据平台往往支撑着公司的搜索、推荐、广告等核心业务，为了保障良好的用户体验和业务效果，运维工作显得十分艰巨。相比于传统的运维方式，大数据时代的运维面临着集群规模更大、业务组件更多、监控可视化与智能化更为复杂等诸多难题。

我们知道，在互联网初期，大部分应用程序跑在少量的服务器上，网络带宽很小，存储量也很小，这个时候的运维更多的是解决类似于组网、操作系统等机房建设问题，应用的上线部署可以由开发工程师来完成，运维的工作职责没有那么明显。随后互联网进入高速发展期，数据规模从 GB 到 TB 再到 PB 级别，在存储量上超过千倍增长，在计算规模上可能也远远超过千倍增长，传统的通过单节点来存储和计算超过 PB 级别的数据已经比较困难，分布式集群的方式已经成为标准的解决方案。分布式系统在存储上解决了大规模数据单机无法承载的问题，同时在计算上解决了单机 CPU 或者内存等资源无法完全满足的问题，但是同时也带来了许多运维难题，诸如统一上线部署、大规模机器管理、降级、容灾、数据同步等。从数据规模到机器规模的扩大，传统的运维方式和方法已经不能满足产品快速迭代的要求，智能运维在这样的场景下应运而生。

智能运维是建立在运维基础上，通过一定策略和算法来进行智能化诊断决策，以更快、更准确、更高效地完成运维工作的技术体系。要实现智能运维的目标，需要有平台支撑，这也是 DevOps 很火的原因，很多运维工程师都掌握了开发工具和平台的本领，因此建立了高效的自动化运维平台。所以说智能运维是运维发展的高级阶段，也是互联网时代发展到一定阶段的产物。

智能运维的基础是建立在大规模数据分析和计算之上，当数据量很小时，我们甚至可以人

工判断和决策，一旦数据达到一定规模，大数据涉及的所有技术就都会成为智能运维所依赖的技术。一方面，可以说智能运维是一种新型技术，因为它从另一个视角去看待运维，对传统运维进行了创新和升华；另一方面，也可以说智能运维是一种经典技术，它是一系列成熟技术的结合体，它融入了运维技术、大数据、传统机器学习技术、机器学习、深度学习等方方面面的技术。

那么在大数据时代应该如何做好运维？我觉得有三个方面。

一是基础设施平台化，大数据的 4V 特性，相比于传统的系统运维，数据的处理框架变得更为多样化和复杂化，这要求我们必须夯实基础设施才能事半功倍。比如多源异构海量数据的分布式存储、离线批处理、高性能索引、大规模流数据处理，以及可视化监控与报警平台等。

二是集群管理自动化，降低运维复杂度。自动化能够提升稳定性，固化的操作交给机器去做，可以降低人为操作失误，提高线上的稳定性；自动化还能极大地提高效率，将运维人员从日常烦琐的操作中解放出来，把更多的时间投入到运维平台迭代优化上，从而更好地为业务运营服务。

三是运维决策智能化，充分利用大数据分析技术提升预测、发现和自动检测的能力，预测分配资源，动态伸缩集群，实现智能预警，自动修复，最大化利用资源，减少开销。

本书作者是微博广告平台资深架构师，从 0 到 1 架构了微博广告大数据运维体系，历经纯人工操作、平台化、自动化、智能化多个发展阶段，可以说踩过许多坑，也总结出一套行之有效的方法论。微博广告五年来一直高速发展而系统始终保持高效稳定，这跟大数据运维体系的成功是分不开的。本书对于即将投入大数据运维行业的工程人员来说是一个非常好的参考，同时对于互联网企业的中高级从业人员也很有借鉴意义。

李东升

微博广告业务部总监

推荐序 4：智能运维也能“开箱即用”

可能鲜为人知的是，智能运维的概念最早并非 Gartner 提出的。

2015 年，国内运维社区热火朝天，也出现了四大运维“门派”。当时王津银同学提出精益运维，智锦同学提出白盒运维，本人萧田国提出高效运维，腾讯游戏的刘栖铜及同仁提出了智能运维。甚至，在上海举行的 2015 年首届 GOPS 全球运维大会，就是运维四大“门派”的集中会演。好不热闹。当年，自动化运维还并非广为人知，由此看来，腾讯游戏确实很有前瞻性。

2016 年，Gartner 提出了 AIOps，基于算法的 IT 运维（Algorithmic IT Operations）。这时也并非智能运维。只是到了 2017 年，Gartner 才修正为 Artificial Intelligence Operations，即目前被广泛接受的人工智能运维。

为什么要回顾这段刚刚发生的历史？这个其实也是我们国人很自豪的地方，也就是说，基于海量业务场景，国内的运维技术及能力可能不仅仅没落后于国外，而且甚至领先于国外。据我所知，截至目前，Facebook 聚焦在将 AI 应用于广告营收，但尚无 AIOps 相关应用。

AIOps，将 AI 这样“高大上”的名词和 Ops 结合到一起，很是美妙。这对于运维而言，可谓一次重生（甚至说，野百合也会有春天）。本来处于软件生产链最末端的、容易被“鄙视”、因为远离业务部门而老背锅的运维同仁，有机会借助 AIOps，从运维升格为技术运营，通过改善产品的用户体验、日活、DAU 及营收和利润，从而和业务部门建立广泛而又建设性的合作，彻底甩掉背锅的命运。

AIOps 可能更“招人待见”（虽然广义来说，AIOps 是 DevOps 在 Ops 侧的高阶实现）。DevOps，从名词上看，容易被人认为是 Dev 把 Ops 吃掉了，而且国内众多企业因为组织结构或监管的要求，难以全面实践 DevOps。但对于 AIOps，运维部门或数据中心可以在自己的能力域竖井式发展，是一种既有面子又有里子的好事情。

那么，互联网中小企业及广大传统企业如何践行 AIOps？毕竟 AIOps 容易让人感觉是阳春白雪，有些高山仰止。

为了集中解决这个问题，为了让广大企事业单位能充分享受到大企业的红利，在中国信息通信研究院指导下，由高效运维社区牵头，邀请到国内互联网、金融及通信行业顶尖企事业单位的 AIOps 负责人，一起制定了国内外首个 AIOps 白皮书《企业级 AIOps 实践建议白皮书》，并在云计算标准及开源推进委员会下正式成立 AIOps 标准工作组，以推进相关标准的制定工作。

AIOps 白皮书的发布在国内引起了很大的轰动，这是凝聚各大 AIOps 顶级专家智慧共识的框架及指南。但美中不足的是，没有具体的实践落地。

本书的出现，很好地解决了上述问题。

我们惊喜地发现，不谋而合的是，新浪微博出品的本书，在整体认知、内容编排及落地实践上，和白皮书基本一致，并可实际指导企业进行 AIOps 平台的建设，可谓相得益彰。

本书同样认为，AIOps 对运维而言是一个新机遇，详细解读了智能运维基础设施，包括开源数据采集技术、分布式消息队列、大数据存储技术、大规模数据离线计算及实时计算、时序数据分析框架等。这些内容和 AIOps 白皮书定义的 AIOps 能力框架如出一辙，但又可实践指导落地，甚至到了 Step By Step 的程度。

本书是关于机器学习、数据聚会与关联技术、数据异常点检测技术、故障诊断和分析策略及趋势预测算法的，更是对 AIOps 白皮书关键技术章节的深度细化和补充。

只有书生才会纸上谈兵。本书更多的是介绍基于新浪微博平台技术部门及广告技术部门的生产实践，最后也很大方地给出了微博广告及平台的智能监控系统实现。

本书有理有据，有说服力。文字历练、表达清晰。相关实践可落地，让有志于实践 AIOps 的企业“开箱即用”，实乃国内运维同仁的又一幸事。故此推荐。

萧田国

高效运维社区发起人、AIOps 标准及白皮书发起人

前言

为什么要写这本书

中国互联网发展非常迅速，一方面得益于互联网基础设施的不断完善；另一方面得益于中国巨大的用户人群和消费市场。网络从 PC 到移动互联网时代过渡非常快，2017 年微博用户中移动端占比已经达到 92%，移动互联网的兴起带来了前所未有的新格局，围绕手机移动端的应用生态逐渐形成，大量 PC 时代的公司已经将产品的主战场转移到移动端。

据中国互联网络信息中心（CNNIC）发布的第 41 次《中国互联网络发展状况统计报告》，截至 2017 年 12 月，中国网民规模达到 7.72 亿，手机网民占比 97.5%，手机支付用户规模增长迅速，达到 5.27 亿，网络直播用户规模达到 4.22 亿，中国拥有人工智能企业 592 家，占全球总数的 23.3%。

互联网尤其是移动互联网的发展，也给企业带来了极大的技术挑战，如何保障线上产品各个服务和系统的稳定性、如何快速高效地诊断问题和定位问题等成为企业所面临的核心问题，而这些问题通过传统的运维方式已经无法有效得到解决。尤其是在大数据复杂场景下，对运维有了新的期待，也对运维有了更高的要求。

首先，系统产生的数据在很大程度上反映了系统状态和产品逻辑，监控系统需要具备快速搜集和处理数据的能力，能经过复杂的数据清洗并从大规模数据中抽取监控需要的指标，尤其是能对时序数据进行 ETL 及存储分析，将异构数据转化成监控系统能够理解的结构化数据。在这个阶段，如何保证数据的一致性和准确性、如何降低时延提高数据吞吐、如何降低监控系统对业务资源的影响等，都是在大数据环境下要解决的问题。

其次，监控系统要与报警系统融合，报警系统承载的是系统风险提示，对准确率要求很高，然而在现实环境下，大部分报警系统都存在报警项繁多、报警次数频繁、报警不准确等问题。

再次，智能化故障诊断、异常点检测、根因分析等是智能运维要解决的核心问题，然而智

能化系统要建立在高效的平台化基础上，目前极大一部分公司还没有真正进入平台化阶段，这就为智能运维带来了极大的阻力。

最后，智能运维需要运维工程师具备一定的人工智能、机器学习及深度学习等算法和建模能力，然而就目前来讲，运维工程师在这方面的技能是比较欠缺的。

国内在智能运维技术上，百度、阿里巴巴、腾讯和微博都有相当程度的经验积累，也得益于这些企业的运维平台化的体系逐渐成熟，在跟这些团队交流的时候，大家都有一个共识，就是希望能将智能运维技术应用到运维的各个维度，也相信智能运维将彻底改变运维的现有体系，并将极大丰富和完善传统运维，提高运维效率。

目前市面上与运维相关的书籍更多的是介绍某个单一技术的运维方法，少有对智能运维进行全面介绍的书籍，因此，我们觉得非常有必要梳理编写一本大数据场景下的智能运维技术的书籍，全面完整地为大家介绍智能运维的技术体系，以及在大企业的运维实践经验，让读者更加了解运维的技术方向，在实践中能够有所借鉴。同时，也能帮助运维工程师在一定程度上了解机器学习的常见算法模型，以及如何将它们应用到运维工作中。

读者对象

本书面向的读者主要包括：

- 运维工程师
- 运维开发工程师
- 运维架构师
- 大数据工程师
- 对运维和大数据，以及 AIOps 感兴趣的工程师

如何阅读这本书

本书主要分 4 篇：第 1 篇运维发展史，重点阐述当前运维的发展现状及面临的技术挑战。第 2 篇智能运维基础设施，重点讲述大数据场景下的数据存储、大数据处理和分析的方法与经验，以及海量数据多维度多指标的处理分析技术。第 3 篇智能运维技术详解，重点关注在新时

期大数据时代下智能化的运维技术，包括数据聚合与关联、数据异常点检测、故障诊断和分析、趋势预测算法；第4篇技术案例详解，为大家梳理了通过开源框架 ELK 快速构建智能监控系统的整体方案，还将分享微博平台和微博广告两个不同业务场景下智能监控系统的技术实践。

具体而言，每篇和章节的主要内容如下：

第1篇 开门见山：运维发展史

- 第1章运维现状，主要介绍运维职责、传统运维、运维分类等，我们将从运维发展的四个阶段：人工、工具和自动化、平台化、智能化来介绍运维的发展现状。
- 第2章智能运维，主要介绍运维面临的挑战，传统运维在海量数据存储、分析、处理，多维度多指标及复杂业务等方面都有一定的局限性，随着运维新时代的到来，智能运维（AIOps）将为运维带来新的机会。

第2篇 站在巨人肩上：智能运维基础设施

- 第3章开源数据采集技术，重点介绍 Filebeat、Logstash 等开源数据采集工具。
- 第4章分布式消息队列，重点介绍以 Kafka 为代表的分布式消息队列及相关技术。
- 第5章大数据存储技术，重点介绍大数据的存储相关技术，这也是大数据场景下的智能运维基础。
- 第6章大规模数据离线计算分析，通过大数据 ETL 技术、Hadoop 技术生态讨论在大数据场景下如何进行离线计算和分析。
- 第7章实时计算框架，重点介绍在实时流计算方面的相关技术和框架，将探讨如何在监控系统中选择和使用实时计算框架。
- 第8章时序数据分析框架，实时监控系統处理的是时序数据，本章介绍常见的时序数据分析框架及使用方法。
- 第9章机器学习框架，智能化是运维、大数据和 AI 的结合，本章简单介绍机器学习框架，并以 TensorFlow 为例介绍如何进行模型训练和实践。

第3篇 运维新时代：智能运维技术详解

- 第10章数据聚合与关联技术，在数据聚合方面讨论聚合方法、多维度数据聚合技巧，以及如何降低维度；在数据关联方面介绍如何在实时流场景下进行时序数据的关联。

- 第 11 章数据异常点检测技术，本章结合运维面临的如异常点检测、动态阈值等常见问题，共同讨论解决这些问题的一些相关模型和算法。
- 第 12 章故障诊断和分析策略，故障诊断是智能运维的一个很重要的研究方向，本章讨论智能运维在故障诊断、决策树模型、关联分析等方面的策略和模型。
- 第 13 章趋势预测算法，主要介绍走势/趋势预测方面的常见模型和方法，包括 ARIMA 及基于机器学习的 LSTM 预测技术。

第 4 篇 智能运维架构实践：技术案例详解

- 第 14 章快速构建日志监控系统，以 ELK 为例介绍如何使用开源框架快速搭建日志监控系统。
- 第 15 章微博广告智能监控系统，全面介绍微博广告智能监控系统架构和设计原理。
- 第 16 章微博平台通用监控系统，以微博平台监控系统为例，全面介绍通用监控系统的设计思路 and 具体架构。

需要注意的是，本文提到的智能运维即指 AIOps，后续篇章将不再进行说明和区分。

勘误和支持

由于笔者的水平有限，编写时间仓促，同时本书在创作过程中参考了大量的国内外技术，并结合实践经验进行了系统性总结。由于技术的发展非常迅速，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。

智能运维技术在发生着翻天覆地的变化，我们希望更多的人能参与到这个过程中，共同推动智能运维技术的普及，欢迎通过微信或者邮件与我们进行讨论。你可以通过微信 justAStriver、微博@AndrewPD 或者电子邮件 contact@andrewpd.com 联系到我们，期待能够得到你们的真挚反馈，在技术之路上互勉共进。

特别致谢

我们花费了大量的时间总结智能运维方面的技术并整理成书，在此非常感谢微博广告基础架构团队的各位同事，尤其感谢车亚强、刘俊、陆松林、王莉、朱伟（按姓氏拼音排序）等人

的辛勤付出，他们在工作之余，挤出宝贵的时间为本书贡献了知识，共同完成了本书部分章节的内容梳理。感谢车亚强在实时流计算的基本概念和关键技术方面，尤其是对分布式消息队列和 Spark 相关技术贡献的内容；感谢刘俊对智能运维的全面介绍，以及在故障诊断技术方面的贡献，同时结合微博平台的应用场景整理了通用监控系统的设计方案；感谢陆松林在分布式存储和离线计算方面提供的案例；感谢王莉在预估模型及异常点检测模型方面的研究；感谢朱伟在运维及智能运维相关技术上的研究和内容贡献。

感谢李东升的大力支持和鼓励，感谢微博广告团队的各位同事、朋友的支持和帮助。

感谢张志强和 TimYang 两位老大抽出宝贵时间为本书写序，感谢裴丹博士、王鹏云、梁定安、饶琛琳、钟华、陈晓峰、陆沛等业界朋友，感谢大家一起推动智能运维行业的普及和发展。

感谢电子工业出版社的编辑张春雨，他的敬业精神令我由衷敬佩，他的反馈、建议、鼓励和帮助引导我们克服诸多困难完成全部书稿。

在此特别感谢我的父母对我的培养，感谢我的太太 Kathy 长期以来对我的默默支持，感谢我刚满 1 岁的女儿小洋葱，因为工作和写作牺牲了很多陪伴她的时间。

技术交流

智能运维技术热潮刚刚开始，希望读者朋友能够跟我们一起多多交流，共同推动中国智能运维技术的发展。你可以通过以下方式联系到我们。

- (1) 微信: justAStriver
- (2) 微博: @AndrewPD
- (3) GitHub: <https://github.com/justastriver>
- (4) 邮箱: contact@andrewpd.com



目录

第 1 篇 开门见山：运维发展史

第 1 章 运维现状.....	2
1.1 运维工程.....	2
1.1.1 认识运维.....	2
1.1.2 主要职责.....	4
1.1.3 运维技术.....	5
1.2 运维发展历程.....	6
1.2.1 人工阶段.....	6
1.2.2 工具和自动化阶段.....	7
1.2.3 平台化阶段.....	7
1.2.4 智能运维阶段.....	8
1.3 运维现状.....	9
1.3.1 故障频发.....	9
1.3.2 系统复杂性.....	10
1.3.3 大数据环境.....	12
1.4 本章小结.....	14
1.5 参考文献.....	14
第 2 章 智能运维.....	15
2.1 海量数据的存储、分析和处理.....	16
2.2 多维度、多数据源.....	18
2.3 信息过载.....	19
2.4 复杂业务模型下的故障定位.....	21
2.5 本章小结.....	22
2.6 参考文献.....	22

第 2 篇 站在巨人肩上：智能运维基础设施

第 3 章 开源数据采集技术.....	25
3.1 数据采集工具对比.....	25

3.2	轻量级采集工具 Filebeat	26
3.2.1	Filebeat 工作原理	26
3.2.2	Filebeat 的安装与配置	28
3.2.3	启动和运行 Filebeat	38
3.3	日志采集解析工具	38
3.3.1	Logstash 工作原理	39
3.3.2	安装 Logstash	40
3.3.3	配置 Logstash	41
3.3.4	启动 Logstash	49
3.4	本章小结	49
3.5	参考文献	50
第 4 章	分布式消息队列	51
4.1	开源消息队列对比与分析	51
4.1.1	概述	51
4.1.2	ZeroMQ	51
4.1.3	ActiveMQ	52
4.1.4	RocketMQ	52
4.1.5	Kafka	53
4.2	Kafka 的安装与使用	53
4.2.1	组件概念	53
4.2.2	基本特性	53
4.2.3	安装与使用	54
4.2.4	Java API 的使用	55
4.3	案例分析	57
4.3.1	日志采集	58
4.3.2	实时结算	58
4.3.3	实时计算	58
4.4	本章小结	58
4.5	参考文献	59
第 5 章	大数据存储技术	60
5.1	传统数据存储	60
5.1.1	传统应用的架构	60
5.1.2	传统存储的运行机制	61
5.1.3	传统存储带来的问题	62
5.2	基于 HDFS 的分布式存储	63
5.2.1	分布式存储的定义	63
5.2.2	HDFS 的基本原理	64
5.2.3	HDFS 架构解析	65
5.2.4	HDFS 的优势	66

5.2.5 HDFS 不适合的场景	67
5.3 分层存储	68
5.3.1 数据仓库	68
5.3.2 数据仓库分层架构	70
5.3.3 分层存储的好处	73
5.4 案例分析	73
5.4.1 数据存储架构	73
5.4.2 数据仓库建模	74
5.4.3 常见的存储问题及解决方案	80
5.5 本章小结	80
5.6 参考文献	80
第 6 章 大规模数据离线计算分析	82
6.1 经典的离线计算	82
6.1.1 Linux 神级工具 sed 和 awk	82
6.1.2 Python 数据处理 Pandas 基础	84
6.1.3 Python 的优势和不足	88
6.2 分布式离线计算	89
6.2.1 MapReduce 离线计算	89
6.2.2 离线计算的数据倾斜问题	97
6.2.3 分布式离线计算的技术栈	100
6.3 案例分析	101
6.3.1 离线计算管理	102
6.3.2 离线计算原子控制	103
6.3.3 离线计算的数据质量	103
6.4 本章小结	104
6.5 参考文献	105
第 7 章 实时计算框架	106
7.1 关于实时流计算	106
7.1.1 如何提高实时流计算的实时性	106
7.1.2 如何提高实时流计算结果的准确性	107
7.1.3 如何提高实时流计算结果的响应速度	107
7.2 Spark Streaming 计算框架介绍	107
7.2.1 概述	107
7.2.2 基本概念	108
7.2.3 运行原理	108
7.2.4 编程模型	109
7.2.5 Spark Streaming 的使用	110
7.2.6 优化运行时间	114
7.3 Flink 计算框架	115

7.3.1	基本概念	116
7.3.2	Flink 特点	116
7.3.3	运行原理	118
7.3.4	Java API 的使用	121
7.4	案例分析	124
7.4.1	背景介绍	125
7.4.2	架构设计	126
7.4.3	效果分析	126
7.5	本章小结	126
7.6	参考文献	126
第 8 章	时序数据分析框架	127
8.1	时序数据库简介	127
8.1.1	什么是时序数据库	127
8.1.2	时序数据库的特点	128
8.1.3	时序数据库的对比	130
8.2	时序数据库 Graphite	131
8.2.1	Graphite 简介	131
8.2.2	Graphite 在微博广告监控系统中的应用	137
8.3	多维分析利器 Druid	139
8.3.1	什么是 Druid	139
8.3.2	Druid 架构	140
8.3.3	Druid 在微博广告监控平台中的应用	144
8.4	性能神器 ClickHouse	147
8.4.1	什么是 ClickHouse	147
8.4.2	ClickHouse 的特性	148
8.4.3	ClickHouse 的不足	149
8.4.4	安装配置 ClickHouse	149
8.4.5	表引擎	153
8.4.6	函数支持	157
8.5	本章小结	160
8.6	参考文献	160
第 9 章	机器学习框架	161
9.1	简介	161
9.2	TensorFlow 介绍	162
9.2.1	什么是 TensorFlow	162
9.2.2	下载安装	162
9.2.3	“Hello TensorFlow”示例	166
9.3	TensorFlow 进阶	166
9.3.1	基础理论	167

9.3.2	模型准备	169
9.3.3	训练数据	169
9.3.4	模型训练	171
9.3.5	生成 seq2seq 句子	174
9.3.6	运行演示	175
9.4	本章小结	178
9.5	参考文献	179

第 3 篇 运维新时代：智能运维技术详解

第 10 章	数据聚合与关联技术	182
10.1	数据聚合	182
10.1.1	聚合运算	183
10.1.2	多维度聚合	186
10.2	降低维度	188
10.2.1	将告警聚合成关联“事件”	189
10.2.2	减少误报：告警分类	190
10.3	数据关联	192
10.4	实时数据关联案例	193
10.4.1	设计方案	193
10.4.2	效果	195
10.5	本章小结	195
10.6	参考文献	195
第 11 章	数据异常点检测技术	196
11.1	概述	196
11.2	异常检测方法	198
11.2.1	基于统计模型的异常点检测	199
11.2.2	基于邻近度的异常点检测	202
11.2.3	基于密度的异常点检测	203
11.3	独立森林	204
11.4	本章小结	207
11.5	参考文献	207
第 12 章	故障诊断和分析策略	208
12.1	日志标准化	209
12.2	全链路追踪	210
12.3	SLA 的统一	210
12.4	传统的故障定位方法	211
12.4.1	监报告警型	211
12.4.2	日志分析型	212

12.5	人工智能在故障定位领域的应用	213
12.5.1	基于关联规则的相关性分析	213
12.5.2	基于决策树的故障诊断	217
12.6	本章小结	222
12.7	参考文献	222
第 13 章	趋势预测算法	223
13.1	移动平均法	223
13.2	指数平滑法	224
13.3	ARIMA 模型	226
13.3.1	简介	226
13.3.2	重要概念	226
13.3.3	参数解释	228
13.3.4	建模步骤	230
13.3.5	ARIMA 模型案例	232
13.4	神经网络模型	236
13.4.1	卷积神经网络	236
13.4.2	循环神经网络	238
13.4.3	长短期记忆网络	239
13.4.4	应用说明	241
13.5	本章小结	241
13.6	参考文献	242

第 4 篇 智能运维架构实践：技术案例详解

第 14 章	快速构建日志监控系统	244
14.1	Elasticsearch 分布式搜索引擎	244
14.1.1	基本概念	244
14.1.2	分布式文档存储与读取	248
14.1.3	分布式文档检索	250
14.1.4	分片管理	252
14.1.5	路由策略	254
14.1.6	映射	255
14.2	可视化工具 Kibana	258
14.2.1	Management	260
14.2.2	Discover	260
14.2.3	Visualize	262
14.2.4	Dashboard	263
14.2.5	Timelion	263

14.2.6	Dev Tools	264
14.3	ELK 搭建实践	265
14.3.1	Logstash 安装配置	265
14.3.2	Elasticsearch 集群安装配置	266
14.3.3	Kibana 安装配置	273
14.4	本章小结	274
14.5	参考文献	274
第 15 章	微博广告智能监控系统	275
15.1	背景介绍	275
15.1.1	监控指标体系	275
15.1.2	功能设计原则	276
15.2	整体架构	277
15.3	核心功能分析	278
15.3.1	全景监控	278
15.3.2	趋势预测	281
15.3.3	动态阈值	285
15.3.4	服务治理	285
15.4	本章小结	287
第 16 章	微博平台通用监控系统	288
16.1	背景	289
16.2	整体架构	290
16.3	核心模块	291
16.3.1	数据采集 (Logtailer)	291
16.3.2	数据路由 (Statsd-proxy)	293
16.3.3	聚合运算 (Statsd)	294
16.3.4	数据分发 (C-Relay) 和数据存储	294
16.3.5	告警模块	295
16.3.6	API 设计	299
16.3.7	数据可视化	300
16.4	第三方应用	301
16.4.1	决策支持系统	301
16.4.2	运维自动化	302
16.4.3	成本分析和容量日报	302
16.4.4	机器学习	302
16.5	本章小节	302
附录 A	中国大数据技术大会 2017 (BDTC 2017) CSDN 专访实录	303

第 1 篇

开门见山：运维发展史

中国互联网从 20 世纪 90 年代开始形成，随即进入了快速发展阶段。第一阶段是以新浪、搜狐和网易为代表的门户网站时期，解决了信息尤其是新闻信息的传播问题。第二阶段是以腾讯、百度和阿里巴巴为代表的科技型时代，解决了社交、信息获取及电商等需求问题。第三阶段进入一个新的时期，从直播、网红到短视频类公司，以及人工智能公司的兴起，足以说明要解决的是心理需求。短短二十年，中国互联网发生了翻天覆地的变化，网民数量已经增长到 9 亿，其中使用手机的用户量已经达到 7 亿，中国也被称为具有全球最大消费能力的国家，很多企业只要做好中国市场，就能与其他做全球市场的企业比肩。科技的进步需要技术的支撑，运维技术也在不断地发生着变化。

运维的技术体系伴随着互联网的发展而愈加完善，从最开始的纯人工运维到脚本和开源工具的使用，再到平台建设，运维也紧跟时代的步伐，逐渐向智能化方向发展，通过应用机器学习的一些算法和模型将极大简化运维对数据的理解，大幅度提升效率。与此同时，运维工作也面临着非常大的挑战，在大数据场景下信息过载、多维度和更加复杂的业务环境，以及故障的定位、检测等工作，都已经超过了单凭人工能够完成的极限，新的技术需要被引进和普及。

近年来，人工智能技术备受关注，开始将 AI 引入 IT 运维领域，AIOps 的概念由此应运而生。Gartner 的报告宣称，到 2020 年，将近 50% 的企业将会在他们的业务和 IT 运维方面采用 AIOps，智能化已是大势所趋。

本篇主要展现运维的发展历史和经历的不同历史阶段，以及运维工作的现状。本篇主要分两个章节：

- 第 1 章 运维现状

- 第 2 章 智能运维

第 1 章

运维现状

1.1 运维工程

1.1.1 认识运维

运维，通常指 IT 运维（IT Operations），是指通过一系列步骤和方法，管理与维护线上服务（Online Service）或者产品（Product）的过程。运维有着非常广泛的定义，在不同的公司不同的阶段代表不同的职责与定位，没有一个统一的标准。尤其是随着互联网的发展，运维的含义也在逐渐互联网化。互联网运维通常属于技术部门，与研发、测试、系统管理同为互联网产品的技术支撑，这种划分在国内和国外以及大小公司之间都会多少有一些不同。

运维的重点在于系统运行的各种环境，从机房、网络、存储、物理机、虚拟机这些基础的架构，到数据库、中间件平台、云平台、大数据平台，偏重的也不是编程，而是对这类平台的使用和管理。运维的水平可以成为衡量一个公司（IT 公司）技术实力的标准。

运维工程师（Operation Engineer），是指从事运维工作的工程师。运维工程师的工作范围非常广泛，包括服务器购买、租用和上架等基本管理，调整网络设备的配置管理和部署，服务器操作系统安装调试，测试环境和生产环境的初始化与维护，代码部署和管理（Git 和 SVN 等），设计和部署线上服务的监控与报警，服务安全性检测（防止漏洞和攻击），数据库管理和调优等。在大型公司中，运维工程师根据工作内容被细化为网站运维、系统运维、网络运维、数据库运维（DBA）、IT 运维、运维开发（DevOps^[1]）、运维安全等方向。

运维的工作内容决定了对运维工程师的要求会非常高，运维工程师需要对服务器资源（CPU、内存、磁盘、网络 IO 等）非常了解，对 Linux 系统和常见的开源框架、工具非常熟悉，因此在

运维工程师中更容易诞生架构师，因为他们知道如何优化服务、如何使得资源利用最大化。

运维工程在国内也被称作 SRE^[2]（Site Reliability Engineering，来自 Google），直接翻译为网站可用性工程。SRE 工程师需要具备算法、数据结构、编程能力、网络编程、分布式系统、可扩展架构、故障排除等各方面技能，其核心工作包括容量规划与实施、服务集群维护、系统容错管理、负载均衡、监控系统以及值班等，最终为产品上线后服务的稳定性负责，但是不负责具体的机器运维。

SRE 工程师的首要工作任务是保障 SLA（Service-Level Agreement，服务等级协议），它定义了对服务有效性的保障，比如对故障解决时间、服务超时等的保障。根据这个协议标准可以定义系统的可用性（Availability），这里需要掌握如下几个衡量指标。

1. 平均故障间隔时间（MTBF）

平均故障间隔时间（MTBF，Mean Time Between Failure），指相邻两次故障之间的平均工作时间。MTBF 通常是衡量一个产品可靠性的指标，这个间隔时间越短说明系统可靠性越差。

2. 平均修复时间（MTTR）

平均修复时间（MTTR，Mean Time To Repair），指产品由故障状态转为工作状态时修复时间的平均值，即故障修复所需要的平均时间。MTTR 值越低说明故障修复越及时。

3. 可用性（Availability）

可用性是系统架构设计中很重要的衡量指标。根据 GB/T3187—97 对可用性的定义，可用性是指在要求的外部资源得到保证的前提下，产品在规定的条件下和规定的时刻或时间区间内处于可执行规定功能状态的能力。它是产品可靠性、维修性和维修保障性的综合反映，如公式（1-1）所示。

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \tag{1-1}$$

业界一般通过 N 个 9 来对可用性进行量化。如表 1-1 所示为对可用性的量化及描述。

表 1-1 系统可用性量化指标

可用性级别	通俗叫法	停机时间/年	描述
99%	2 个 9	87.6 小时	基本可用性
99.9%	3 个 9	8.8 小时	较高可用性
99.99%	4 个 9	53 分钟	具有故障自动恢复能力的可用性

续表

可用性级别	通俗叫法	停机时间/年	描述
99.999%	5 个 9	5 分钟	极高可用性
99.9999%	6 个 9	31 秒	

可以看出，当系统可用性超过 3 个 9 时，全年停机状态持续时间低于 8.8 小时，此时一般可以称作高可用性系统，可用性越高，对系统的设计要求就越高。需要注意的是，在分布式系统中，可用性和性能应该遵循 CAP 原则，即数据一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）三者通常只能满足其中两者，无法兼得。

1.1.2 主要职责

运维的主要职责是在产品生命周期的各个阶段，维护系统的稳定性。运维的职责覆盖了产品从设计到发布、运行维护、变更升级及至下线的生命周期，在产品生产环境各个阶段的职责也不同，如表 1-2 所示。

表 1-2 运维在产品研发的生命周期中的主要职责

产品阶段	运维职责
设计阶段	<ul style="list-style-type: none">稳定性评估：主要是针对系统架构设计的合理性进行评估，包括是否存在单点、是否可容错、是否有强耦合等，同时评估能够满足上线发布并稳定运行的基本要求资源评估：包括对所需的服务器资源、网络资源以及资源的分布等进行评估，同时把握相关产品对资源预算申请的合理性，控制资源使用成本资源申请和准备
开发阶段	环境部署、依赖库及包管理、操作系统维护、数据库准备等
测试阶段	测试环境部署，稳定性评估，从系统的稳定性和可运维性的角度提出开发需求
部署阶段	自动化部署、稳定性检验、可扩展部署等
线上运行阶段	<p>保证线上服务的稳定运行：</p> <ul style="list-style-type: none">实时监控：对服务运行的状态进行实时监控，随时发现服务的运行异常和资源消耗情况；输出重要的日常服务运行报表，以评估服务/业务整体运行状况，发现服务隐患故障处理：对服务出现的任何异常进行及时处理，尽可能避免问题扩大化甚至中止服务容量管理：包括服务规模扩张后的资源评估、扩容、机房迁移、流量调度等规划和具体实施
下线/回滚阶段	由于产品效果不如预期或者其他原因，产品可能需要做下线或者回滚处理，在这个过程中运维工程师主要做好资源回收的工作，将服务终止，对机器/网络等资源进行回收

可以看出,运维工作贯穿于产品研发的各个环节。因此,运维工程师的日常工作主要包括:

- 产品技术方案评估。
- 资源预估、申请和管理。
- 环境部署、环境准备。
- 产品上线、下线及回滚,服务在线发布/升级产品。
- 监控线上的服务质量。
- 响应异常/处理突发故障。
- 和相应产品线的研发和测试团队协调处理产品问题。
- 与产品、技术、测试等团队沟通协作。
- 对系统实时数据进行分析。
- 建立工具或平台,保障系统的稳定性和可靠性。

1.1.3 运维技术

在产品的整个生命周期中运维工作重要而广泛,但运维工程师的职责不局限于这部分工作,还需要总结工作中遇到的问题,抽取出相关的技术方向,研发相关的工具和平台,以支持/优化业务的发展并提高运维的效率。

运维工程师所需的技术体系根据其专业方向而异,但对计算机系统架构、操作系统、网络技术的掌握是基本要求。例如,可能需要熟练掌握 Linux 操作系统的使用,熟练使用各种脚本工具来处理日常工作任务,精通 TCP/IP 协议栈以排查一个大规模网络系统中的流量异常问题等。更进一步的,需要形成一套软件可运维性方面的经验,以此作为后续工作的指导。

一个运维工程师在初期阶段的目的是掌握、维护一套系统所需的所有软硬件知识和经验。在进阶阶段需要能够设计开发一套基础的体系软件,以支撑业务系统的稳定可靠运行,即开发服务于软件的软件,以支持更大规模的业务系统,提高运维生产力。在最高阶段要能反作用于软件系统的构建和运行阶段,使得系统从诞生阶段起即具有天然的可运维性,以最大化系统的生产力,同时最小化对外部支撑资源的依赖。

运维以技术为基础,通过技术保障产品提供更高质量的服务。运维工作的职责及在业务中的位置决定了运维工程师需要具备更加广博的知识和深入的技术能力,需要掌握的技术非常广

泛，表 1-3 中列出了一些运维涉及的常见技术和框架。

表 1-3 运维涉及的常见技术和框架

功能描述	技术和框架
操作系统	Linux、Ubuntu、Windows、CentOS、Redhat 等
Web Server	以 Nginx 为典型代表，以及 Apache 等
网络工具	tcpcopy、curl 等
监控和报警系统	Grafana、Zabbix 等
自动部署	Ansible、sshpt、salt、Jenkins 等
配置管理及服务发现	Puppet、Consul、Zookeeper 等
负载均衡	LVS、HAProxy、Nginx 等
传输工具	Scribe、Flume 等
集群管理工具	Zookeeper 等
数据库	MySQL、Oracle、SQL Server 等
缓存技术（Cache）	Redis、Memcache 等
消息队列	Kafka、ZeroMQ 等
大数据平台	HDFS、MapReduce、Spark、Storm、Hive 等
大数据存储	HBase、Cassandra、MongoDB、LevelDB 等
时序数据（OLAP 平台）	Druid、OpenTSDB 等
容器	LXC、Docker、K8s 等
虚拟化	OpenStack、Xen、KVM 等

1.2 运维发展历程

伴随着互联网、移动互联网的发展，运维的发展大致经历了如下四个重要阶段。

1.2.1 人工阶段

早期运维处于人工阶段，这个时候的运维是一个通称，负责从机房、服务器选型，软硬件初始化，服务上下线，配置监控，盯监控等，运维和开发之间没有太明确的分工，基本是遇到什么问题解决什么问题。在这个阶段，因为服务器少、业务需求简单，只需要少数几个运维工程师就能完成运维工作，运维也基本不会成为企业发展的瓶颈。

1.2.2 工具和自动化阶段

通过人工维护和管理的方式，比较适合简单的业务，随着 IT 尤其是互联网的迅速发展，企业所承担的业务愈发复杂，运维也进入了第二阶段，即工具和自动化阶段。为了提升运维工作效率，简化操作流程，运维工程师开始将部分运维操作及重复性工作流程编写成脚本自动执行。

工具的产生是运维自动化的一个典型的标志，尤其是开源项目的逐渐兴起，大量工具被开源，很多项目都可以在 GitHub 上找到。随着容器技术的兴起，许多新的专门运行容器的 Linux 发行版本也出现了。Docker 及相关虚拟化技术的不断发展、K8s（Kubernetes，自动化容器管理工具，具备集群管理、任务调度等强大的功能）等技术的发展、助力云服务（Cloud，包括私有云、公有云）的快速发展，也进一步为运维自动化带来了极大的便利。

在工具和自动化阶段，运维工程师有一部分掌握了一定的开发能力，将日常的工作通过自动执行程序来完成，以替代人工，效率也逐渐比单纯人工运维更高，同时出错的概率逐渐降低。此时已经有掌握 Python、shell 等开发语言和工具的运维工程师逐渐转向运维开发的角色。

1.2.3 平台化阶段

在平台化阶段前，脚本和工具是分散的，不易管理，同时也需要人工干预，随着业务变得更加复杂，对大量脚本的管理是低效和复杂的。之后，将自动化脚本和工具进行整合，从系统层面构建更加易用和高效的运维管理工具，已经成为趋势，这也就是运维平台化。

围绕开源工具、开源平台，大中型企业开始结合业务构建自己的运维平台，包括监控平台、报警平台和自动化平台等，这些平台在一定程度上提高了产品开发效率和测试效率，降低了运维成本，并且降低了系统出风险的概率，提高了系统可用性。

在表 1-4 中简单描述了企业围绕开源框架构建运维体系的各类平台。

表 1-4 运维平台示例

平台类型	典型的基础开源平台	说明
监控系统	ELK、Grafana、OpenTSDB 等	围绕数据搜集、ETL、搜索、指标管理、可视化展示等，快速构建监控系统
报警系统	Zabbix	具有报警管理、报警聚合、提醒等功能，同时与监控系统配合，构成系统稳定性的保障体系
自动化平台	GitLab、Jenkins、Ansible、sshpt、salt、Docker	具有代码托管、编译、打包、环境部署、安装和回滚、灰度等基础功能，同时结合监控和报警系统构建动态扩缩容、自动化降级等系统

可以看出，具有平台化思想的开源工具几乎覆盖了运维的全部维度，从监控系统、报警系统到自动化平台都有非常优秀的开源技术框架。

1.2.4 智能运维阶段

这个阶段是智能化的，当基础设施固定下来后，运维模式最终也会固定下来，这些模式会把可用性、扩容等场景在内的诸多运维方案包含进来，把开发平滑地加入运维架构。AIOps (Algorithmic IT Operations) 最早由 Gartner 定义为采用人工智能算法 (AI 和机器学习)，利用机器解决已知的问题和潜在的运维问题的一种技术解决方案。这里值得注意的是，根据 Gartner 定义，AIOps 中的 AI 并不是 Artificial Intelligence (人工智能)，而是广义的算法。本书中也将遵从此定义，并且不严格区分智能运维与 AIOps。

AIOps 使用分析理论和机器学习等方法，分析和处理各种操作工具、服务和设备产生的大量数据。它能够自动发现问题，并且能够实时对问题做出反应。AIOps 建立在大数据与机器学习 (Machine Learning) 基础之上，如图 1-1 所示。

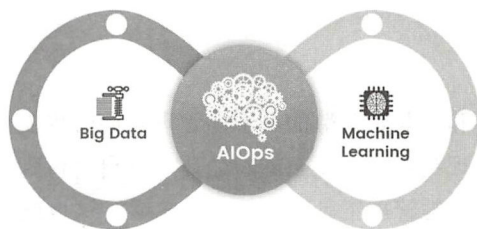


图1-1 基于大数据与机器学习的AIOps

举一个简单的例子。通常的报警策略是设置一个阈值范围（上、下界），当某个指标超出这个阈值范围时，则触发报警。这也是最直接、最简便的办法。然而，对于一些特定场景下的报警，设置可能没有这么简单。如图 1-2 所示是微博广告某产品某天的广告曝光次数走势图，从图中可以看出，每天早上 4 点到 6 点达到一天的最低，上午将近 11 点达到一天的最高。如果依然按照固定的阈值设置报警是非常不准确的，我们需要通过历史数据智能化地拟合出一条趋势线，以这条线的上、下界一定范围作为动态的报警阈值，以达到更加准确的报警。

另外，智能运维可以被用于故障分析，以快速定位问题，降低企业的损失。在 AI 中我们使用到的各类算法，比如基于指数平滑的二次平滑、三次平滑算法，基于 ARIMA 的算法，基于深度学习的前馈神经网络、循环神经网络 (RNN) 算法等，已经比较成熟，并大量使用在其他研究领域，比如图形图像处理、语音识别等领域。所以，在算法上，我们在很早之前就应该具

备了这方面的理论基础。

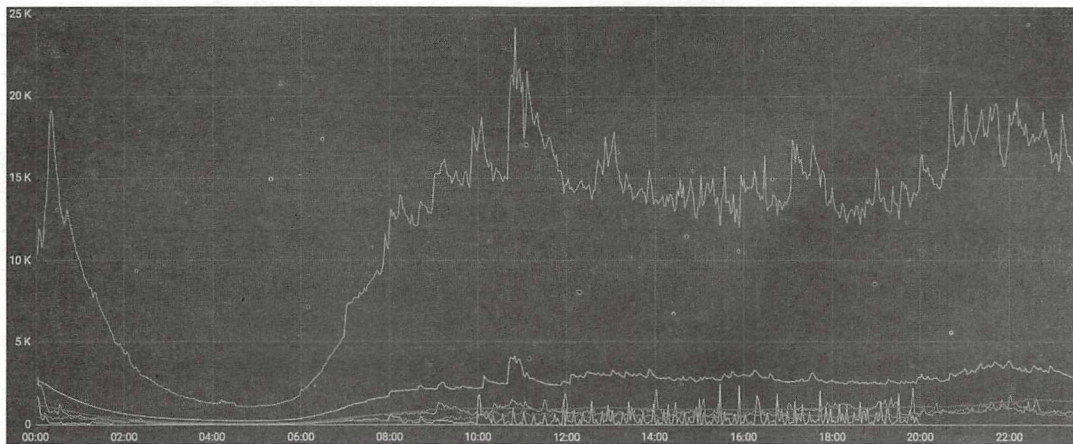


图1-2 微博广告某产品某天的广告曝光次数走势图

在计算能力上，目前我们看到，基于大数据技术的数据处理能力已经足够，Hadoop、Spark等生态，包括时序数据的处理能力，都已经能够支撑智能化的复杂计算场景。

就目前来看，国内的百度、搜狗、宜信、阿里巴巴、微博等都已经探索尝试了 AIOps，并且取得了不错的收益。在 2017 年 InfoQ 举办的 QCon 全球架构大会上，已经有不少与 AIOps 相关的议题，感兴趣的读者可以关注。

1.3 运维现状

1.3.1 故障频发

快速解决故障、降低故障率、不断提高系统可用性是运维的非常关键的职责，然而在大数据场景下，系统越来越复杂，系统维护成本越来越高。Gartner Group 有一个调查显示，在 IT 项目经常出现的问题中，源自技术或产品（包括硬件、软件、网络、电力失常及天灾等）的问题只占 20%，但流程失误方面却占 40%，人员疏失方面也占到了 40%。这些年来，企业通过自动化运维平台，以及 DevOps 等协作理念，逐步解决了 Gartner Group 提到的与流程失误和人员疏失相关的 80% 的问题，但企业系统故障带来的损失也是非常大的。

表 1-5 统计了在 2015 年到 2017 年间，国内外知名互联网企业出现的故障。可以看出，即使是顶级的企业，也面临着各类故障导致的用户流失、财产损失等巨大的风险。

表 1-5 国内外知名互联网企业故障列表（部分）

时间	企业/产品线	故障描述
2015 年	阿里巴巴/支付宝	支付宝因杭州机房网络光纤被挖，导致数小时部分用户业务不可用
	携程	携程网瘫痪事件，全网业务中断 12 小时
	知乎	知乎机房故障，影响系统使用近 2 小时
	阿里云	阿里云香港节点宕机，业务中断 13 小时
	七牛云	七牛云存储服务故障，业务中断 83 分钟
2016 年	GitHub	GitHub 全球服务中断，所有托管在上面的开源项目受到影响，影响时长超过 6 小时
	亚马逊	亚马逊电商网站中断访问，亚马逊电子商务主网站及云计算服务受到影响，持续 20 分钟
	阿里巴巴/支付宝	由于华南一处机房出现故障，支付宝出现故障，无法支付，部分用户无法在线上或线下通过支付宝进行支付购买
	腾讯微信	腾讯微信故障，朋友圈无法打开，微信图文也无法打开，持续 2 小时
	Google	谷歌云存储及文件备份服务器服务中断，部分云用户在访问服务器时会显示“服务器遇到错误，请稍后再试”的提示，持续 20 分钟
2017 年	IBM	2017 年年初，IBM 云的信用度受到影响，客户用于访问其 Bluemix 云基础架构（以前称为 SoftLayer）的一个管理网站服务中断了数小时
	GitLab	GitLab 极受欢迎的线上代码库——GitLab.com 遭遇了 18 小时的服务中断，最终无法完全修复。故障原因是员工在维护过程中从错误的数据库服务器中删除了数据库目录
	亚马逊	亚马逊 RDS 服务上的 MySQL 数据库文件大小限制引发了 Pinterest 服务器的长时间宕机
	亚马逊 AWS	一位 AWS 工程师试图调试亚马逊的弗吉尼亚数据中心 S3 存储系统，但输入了一个错误指令，导致许多互联网——包括 Slack、Quora 和 Trello 等众多企业平台宕机 4 小时
	微软 Azure	微软 Azure 公有云出现超过 8 小时的存储可用性问题，主要影响到美国东部的客户，导致有些用户无法配置新的存储空间或访问本地现有资源。之后，微软工程团队确认原因为断电导致的存储集群不可用
	今日头条	因服务器故障，今日头条全站及头条号后台全部无法访问
	百度	2017 年 2 月 28 日晚，百度出现大规模宕机事件
	新浪微博	新浪微博系统出现故障，网友通过 ping 命令测试 IP 地址的可用性时发现，新浪微博的服务器已经失去响应，此次新浪微博宕机时间接近 1 小时

1.3.2 系统复杂性

当前的 IT 项目基础设施环境与 5 年前已经无法同日而语，更不用说 10 年前了。近几年，

随着云计算、微服务等技术的流行，以及互联网业务的迅速发展，运维人员要关注的服务数量也呈现指数级增长，自动化运维虽然提升了效率，解决了一部分问题，但也遇到了新的难题，比如面对繁多的报警信息，运维人员应该如何处理；当故障发生时，又如何能够迅速定位问题。

系统的复杂性是业务复杂性的结果，我们可能无法直接评估当前互联网企业的系统有多复杂，但可以从客观数据反映出来。

从代码量角度看，Linus Torvalds 最初发布的 Linux 内核版本源码大概 1 万行，而随着操作系统的发展，当前 Linux 内核源码在千万行级别，完整的 Linux 操作系统的代码量过亿行。2015 年 Google 披露全部代码量大概为 20 亿行，按 Google 工程师每天编写 150 行代码估算，每天新增约百万行代码；国内一线企业的代码量也达到 10 亿行级别。这样庞大的代码量，从侧面反映了系统的复杂性。

从产品和业务角度看，仅仅微博广告系统中的业务监控指标就已经超过 10 万个，微博的监控指标量达百万级，这样量级的背后是一套非常庞大的业务系统支撑。BAT（百度、阿里巴巴、腾讯）这样的一线互联网企业，产品线非常多，系统就更加庞大了。如图 1-3 所示是百度首页展示的一部分产品列表，全部产品超过 105 个。



图1-3 百度产品大全（部分）

如图 1-4 所示是某云服务平台包括的产品列表，有超过 100 个复杂的产品服务。这样庞大

的产品系列背后的业务逻辑会非常复杂，对系统稳定性有非常苛刻的要求。

弹性计算	存储和CDN	数据库	网络
云服务器 ECS <small>NEW</small>	对象存储 OSS	云数据库 MySQL 版	专有网络 VPC
轻量应用服务器	块存储	云数据库 SQL Server 版	负载均衡 SLB
弹性裸金属服务器（神龙）	文件存储 NAS	云数据库 Redis 版	NAT 网关
超级计算集群（公测中） <small>NEW</small>	表格存储 TableStore	云数据库 MongoDB 版	弹性公网 IP
GPU 云服务器	归档存储 OAS	云数据库 POLARDB（公测中） <small>NEW</small>	高速通道
FPGA 云服务器（公测中）	云存储网关（公测中）	云数据库 PPAS 版	VPN 网关
块存储	闪存助力	云数据库 PostgreSQL 版	共享流量包
专有网络 VPC	混合云存储阵列	云数据库 Memcache 版	共享带宽
负载均衡 SLB	智能云相册（公测中）	表格存储 TableStore	云解析 PrivateZone <small>NEW</small>
弹性高性能计算 E-HPC	智能媒体管理（公测中） <small>NEW</small>	云数据库 HBase 版	安全加速 SCDN <small>NEW</small>
弹性伸缩	混合云备份服务（公测中） <small>NEW</small>	分布式关系型数据库服务 DRDS	PCDN
资源编排	混合云灾备服务（公测中） <small>NEW</small>	HybridDB for MySQL	CDN
容器服务	安全加速 SCDN <small>NEW</small>	HybridDB for PostgreSQL	大数据基础服务
容器服务 Kubernetes 版（公测中） <small>NEW</small>	PCDN	高性能时间序列数据库 HiTSDB	MaxCompute
容器镜像服务（公测中）	CDN	数据传输 DTS	分析型数据库
批量计算	安全	应用与数据库迁移 ADAM（公测中）	E-MapReduce
函数计算	DDoS 高防 IP	数据管理 DMS	流计算（公测中）
域名与网站	Web 应用防火墙	混合云数据库管理 HDM（公测中） <small>NEW</small>	DataWorks（公测中）
域名注册	游戏盾	数据库备份 DBS（公测中） <small>NEW</small>	数据集成（公测中）
域名交易	云游戏	开放搜索	Dataphin（公测中） <small>NEW</small>
域名抢注 <small>NEW</small>	安全加速 SCDN	Elasticsearch	Elasticsearch <small>NEW</small>
云解析 DNS	安骑士	人工智能 ET	大数据分析 & 展现
HTTPDNS	态势感知	机器学习 PAI	
	CA 证书服务		

图1-4 某云服务平台包括的产品列表（部分）

1.3.3 大数据环境

随着大数据政策环境和技术手段的不断完善，大数据行业应用持续升温，中国企业级大数据市场进入了快速发展时期。互联网、电信、金融等开始实际部署大数据平台并付诸实践，带动了软件、硬件和服务市场的快速发展。

中国信息通信研究院公布的《中国大数据发展调查报告（2017）》显示，2016 年约 70% 的企业拥有的数据资源总量在 50~500TB 之间，18.4% 的企业数据量在 500TB 以上，与 2015 年相比，企业资源总量呈增长趋势。

数据规模的增大，一方面反映了系统的复杂程度；另一方面也反映了监控系统、自动化系统等运维平台的复杂程度。在大数据场景下，运维面临的主要挑战有以下几个方面。

1. 数据采集

数据采集是大数据分析处理的基石，其核心一是要保证数据的完整性；二是要保证数据的准确性；三是要保证数据的实效性。数据完整性要求采集系统能够尽可能搜集到足够多和完整



的信息，在采集过程及预处理过程中都不能丢数据。数据准确性要求在数据采集过程中，不能因为预处理而导致数据不一致，影响后续的分析 and 决策。数据实效性一方面要求数据采集要做到实时或者准实时，采集系统导致的延时率尽可能低，性能尽可能高；另一方面要求在数据预处理阶段，保留数据尤其是时序数据时间效应。这里的时序数据时间效应是指某个指标以某个固定时间间隔的波动变化情况，这个波动在一定程度上反映出系统的运行状态，数据采集器要在系统承载能力允许的前提下缩小时间间隔。比如对于请求量的走势，采集器可以 1 秒、5 秒、15 秒甚至 1 分钟采集一次数据，这样的时间间隔会带来不同的计算误差，最理想的情况是时间间隔越小越好，但带来的问题是数据规模的成倍增长，以及对后续数据分析的极大挑战。

2. 数据存储

目前大数据的原始数据及数据仓库存储介质一般在 HDFS 上，为了提升数据分析能力，部分数据也存储在 HBase、Hive、Redis 等集群上。

各个业务系统不断在产生和制造大量的数据，数据被分析和处理再加工再存储，在每个环节数据都会被复制，一般情况下，原始数据规模最大，越接近数据业务分析层，数据规模越小。

如表 1-6 所示是微博广告某产品线 2016 年数据仓库各层的存储规模。

表 1-6 微博广告某产品线 2016 年数据仓库各层的存储规模

仓库分层类型	每天增量 (TB)
仓库分层——ODS (原始层)	4.053
仓库分层——DWD (明细层)	5
仓库分层——DWS (聚合层)	3
仓库分层——DM (集市层)	2
各层共计	14

如表 1-7 所示是微博广告某产品线存储 1.5 年总体数据规模。

表 1-7 微博广告某产品线存储 1.5 年总体数据规模

类型	数量
存储日增量 (TB)	12.5643
增量系数	1.2
存储时长 (年)	1.5
天数	365



续表

类型	数量
副本数	3
压缩比	0.33
存储量 (TB)	8172

可以看出，数据存储规模是一个非常大的挑战。其中为了保证数据的可用性，一般都会有至少 3 个数据副本（设置 HDFS 副本数是为了保证数据可用性），同时为了节约存储资源，通常采用特定的数据压缩算法来降低存储量。

3. 分析和建模

数据分析和建模体现在数据集的大规模计算上，模型的训练是非常消耗资源的，数据需要在不同的计算节点之间进行复制和传播，都要耗用存储资源和网络带宽，而对数据的处理则需要耗用 CPU 和内存资源。

1.4 本章小结

在大规模、复杂架构的催生下，运维技术不断变化和发展。自动化运维被推到一个新的高度，给传统企业带来了福音，给基础运维带来了巨大的挑战与机遇，同时也给越来越多的企业带来了新的抉择；开源技术的飞跃发展、脚本语言的进化等，也给运维行业带来了革命性的影响。

1.5 参考文献

[1] DevOps 维基百科定义：<https://en.wikipedia.org/wiki/DevOps>

[2] SRE 维基百科定义：https://en.wikipedia.org/wiki/Site_reliability_engineering



第 2 章

智能运维

得益于 IT 外包服务的发达，现在的运维已经不包括搬机器上架、接网线、安装操作系统等基础工作，运维人员一般会从一台已安装好指定版本的操作系统、分配好 IP 地址和账号的服务器入手，工作范围大致包括：服务器管理（操作系统层面，比如重启、下线）、软件包管理、代码上下线、日志管理和分析、监控（区分系统、业务）和告警、流量管理（分发、转移、降级、限流等），以及一些日常的优化、故障排查等。

随着业务的发展、服务器规模的扩大，以及云化（公有云和混合云）、虚拟化的逐步落实，运维工作就扩展到了容量管理、弹性（自动化）扩缩容、安全管理，以及（引入各种容器、开源框架带来的复杂度提高而导致的）故障分析和定位等范围。

听上去每一类工作都不简单。不过，好在这些领域都有成熟的解决方案、开源软件和系统，运维工作的重点就是如何应用好这些工具来解决问题。

传统的运维工作经过不断发展（服务器规模的不断扩大），大致经历了人工、工具和自动化、平台化和智能运维（AIOps）几个阶段。如前文所述，这里的 AIOps 不是指 Artificial Intelligence for IT Operations，而是指 Algorithmic IT Operations（基于 Gartner^[1]的定义标准）。

基于算法的 IT 运维，能利用数据和算法提高运维的自动化程度和效率，比如将其用于告警收敛和合并、Root 分析、关联分析、容量评估、自动扩缩容等运维工作中。

在 Monitoring（监控）、Service Desk（服务台）、Automation（自动化）之上，利用大数据和机器学习持续优化，用机器智能扩展人类的能力极限，这就是智能运维的实质含义。

智能运维具体的落地方式，各团队也都在摸索中，较早见效的是在异常检测、故障分析和定位（有赖于业务系统标准化的推进）等方面的应用，后面的章节会具体涉及。智能运维平台



逻辑架构如图 2-1 所示。

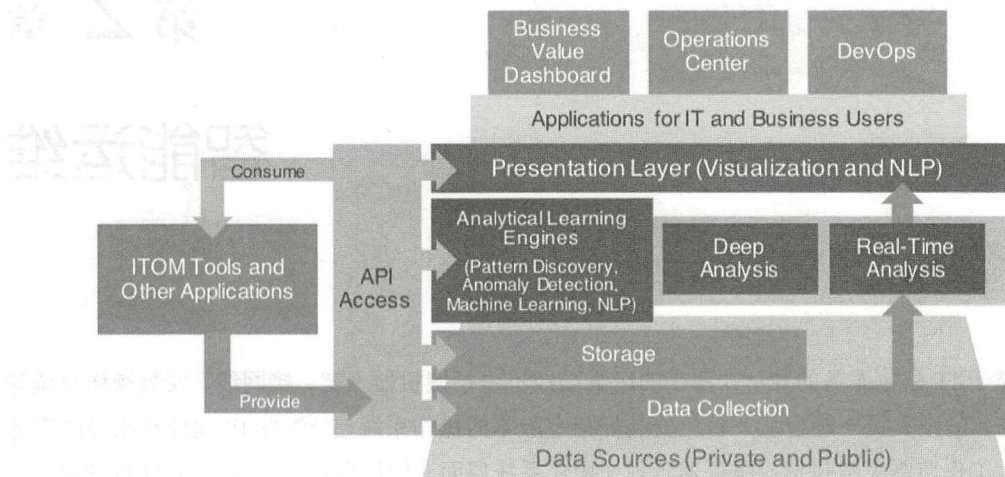


图2-1 智能运维平台逻辑架构图

智能运维决不是一个跳跃发展的过程，而是一个长期演进的系统，其根基还是运维自动化、监控、数据收集、分析和处理等具体的工程。人们很容易忽略智能运维在工程上的投入，认为只要有算法就可以了，其实工程能力和算法能力在这里同样重要。

那么，智能运维在工程方面会有哪些难题呢？这些难题是否会随着智能运维的深入应用而得到一定程度的解决呢？下面的章节会逐步展开这些问题，并提供一些解决方案。

2.1 海量数据的存储、分析和处理

运维人员必须随时掌握服务器的运行状况，除常规的服务器配置、资源占用情况等信息外，业务在运行时会产生大量的日志、异常、告警、状态报告等，我们统称为“事件”。通常每台服务器每个时刻都会产生大量这样的“事件”，在有数万台服务器的场合下，每天产生的“事件”数量是数亿级的，存储量可能是 TB 级别的。

在过去，我们通常采用的方法是将日志保留在本地，当发现问题时，会登录出问题的服务器查看日志、排查故障，通过 sar、dmesg 等工具查看历史状态；监控 Agent 或者脚本也会将部分状态数据汇报到类似于 Zabbix 这样的监控软件中，集中进行监控和告警。

当服务器规模越来越大时，如何统一、自动化处理这些“事件”的需求就越来越强烈，毕



竟登录服务器查看日志这种方式效率很低，而成熟的监控软件（比如 Zabbix、Zenoss 等）只能收集和处理众多“事件”当中的一部分，当服务器数量多了以后，其扩展能力、二次开发能力也非常有限。在具体实践中，当监控指标超过百万级别时，就很少再使用这种单一的解决方案了，而是组合不同的工具和软件，分类解决问题。

在通用设计方法中，有“大工具、小系统，小工具、大系统”的说法，这也符合 UNIX 的设计哲学，每个工具只做好一件事，一堆小工具组合起来可以完成很复杂的工作。如果使用的是一些大工具或者系统，表面上看功能很多，但是当你想处理更复杂的业务时，就会发现每一个功能都不够用，而且还很难扩展，它能做多“大”事取决于它的设计，而不是你的能力。

一个由典型的小工具组成的大系统，任何一个部分都可以被取代，你完全可以用自己更熟悉的工具来做，而且对工具或者组件的替换，对整体没有太大影响。

一提到海量数据的存储、分析和处理，大家就会想到各种各样的大数据平台。是的，大数据平台确实是用来处理海量数据的，但反过来不见得成立，对海量数据的分析和处理，并不总是或者只依赖大数据平台。

“分类”这个词听上去朴实无华，然而处理复杂问题最基本的方法就是分类，甚至“分类方法”也是机器学习非常重要的组成部分。“海量数据处理”这是一个宏大的命题，听上去让人一头雾水，但当你将“事件”或者需要处理的问题分类后，每一部分看上去就是一个可以解决的问题了。

后面的章节会详细介绍如何对海量“事件”进行分类和处理。

- 实时数据和非实时数据。
- 格式化数据和非格式化数据。
- 需要索引的数据和只需要运算的数据。
- 全量数据和抽样数据。
- 可视化数据和告警数据。

每一个分类都对应一种或多种数据处理、分析和存储方式。也可以说，当你对数据、需求完成分类后，基本的框架也就定了下来，剩下的工作就是集成这些工具。



2.2 多维度、多数据源

如图 2-2 所示，这是一个多维度模型示例。真实世界的情况是（至少按弦理论学家所说的是），除我们可以感知的 3 个“延展维”外，还有 6 个“蜷缩维”，它们的尺寸在普朗克长度的数量级，因此我们无法感知到。

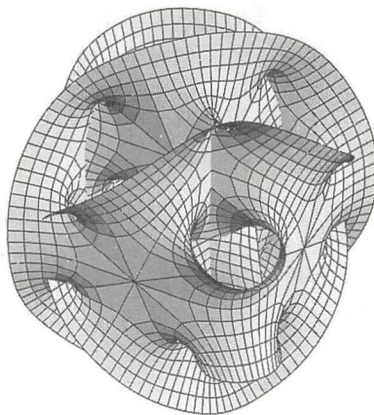


图2-2 多维度模型示例

当然，运维数据中的“多维度”，还没有复杂到这样难以理解。

在相对复杂的业务场景下，一个“事件”除包含我们常用的“时间”（何时发生）、“地点”（哪个服务器/组件）、“内容”（包括错误码、状态值等）外，还应当包含地区、机房、服务池、业务线、服务、接口等，这就是多维度数据。

很多时候，数据分析人员可能要使用各种维度、组合各种指标来生成报告、Dashboard、告警规则等，所以是否支持多维度的数据存储和查询分析，是衡量一个系统是否具有灵活性的重要指标。

对多维度数据的处理，很多时候是一个协议/模型设计问题，甚至都不会牵扯具体的分析和处理框架，设计良好的协议和存储模型，能够兼顾简洁性和多维度。

在后面的章节中，我们会介绍协议/模型设计中的多维度。

- 在单一 Key 中包含维度信息。
- 在 Tag 中标注不同维度。



不同的设计理念会对应不同的处理模型，没有优劣之分，只有哪个更合适的区别。

多数据源或者说异构数据源已经很普遍了，毕竟在复杂场景下并不总是只产生一种类型的数据，也不是所有数据都要用统一的方式处理和存储。

在具体的实践中，通常会混合使用多种存储介质和计算模型。

- 监控数据：时序数据库（RRD、Whisper、TSDB）。
- 告警事件：Redis。
- 分析报表：MySQL。
- 日志检索：Elasticsearch、Hadoop/Hive。

这里列出的只是一部分。

如何从异构的多数据源中获取数据，还要考虑当其中某个数据源失效、服务延迟时，能否不影响整个系统的稳定性。这考量的不仅仅是各种数据格式/API 的适配能力，而且在多依赖系统中快速失败和 SLA 也是要涉及的点。

多数据源还有一个关键问题就是如何做到数据和展现分离。如果展现和数据的契合度太高，那么随便一点变更都会导致前端界面展现部分的更改，带来的工作量可能会非常大，很多烂尾的系统都有这个因素存在的可能性。

2.3 信息过载

DDoS（分布式拒绝服务）攻击，指借助于客户/服务器技术，将多台计算机联合起来作为攻击平台，对一个或多个目标发动攻击。其特点是所有请求都是合法的，但请求量特别大，很快会消耗光计算资源和带宽，图 2-3 展示了一个 DDos 攻击示例。

当我们的大脑在短时间接收到大量的信息，达到了无法及时处理的程度时，实际上就处于“拒绝服务”的状态，尤其是当重大故障发生，各种信息、蜂拥而至的警报同时到达时。

典型的信息过载的场景就是“告警”应用，管理员几乎给所有需要的地方都加上了告警，以为这样即可高枕无忧了



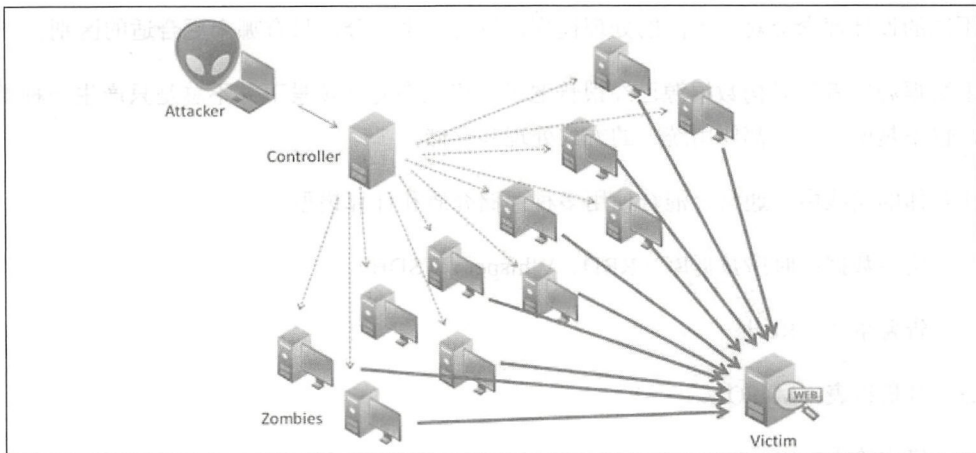


图2-3 DDoS攻击示例

然而，接触过告警的人都知道，邮件、短信、手机推送、不同声音和颜色提醒等各种来源的信息可以轻松挤满你的空间，很多人一天要收上万条告警短信，手机都无法正常使用，更别谈关注故障了。

怎样从成千上万条信息中发现有用的，过滤掉重复的、抖动性的信息，或者从中找出问题根源，从来都不是一件容易的事情，所以业界流传着“监控容易做，告警很难报”的说法。

还有一个场景就是监控，当指标较少、只有数十张 Dashboard 时，尚且可以让服务台 24 小时关注，但是当指标达到百万、千万，Dashboard 达到数万张时（你没看错，是数万张图，得益于 Grafana/Graphite 的灵活性，Dashboard 可以用程序自动产生，无须运维工程师手工配置），就已经无法用人力来解决 Dashboard 的巡检了。

历史的发展总是螺旋上升的，早期我们监控的指标少，对系统的了解不够全面，于是加大力度提高覆盖度，等实现了全面覆盖，又发现信息太多了，人工无法处理，又要想办法降噪、聚合、抽象。少→多→少这一过程看似简单，其实经过了多次迭代和长时间的演化。

在后续章节中，将为读者介绍这类问题在实践中的解决方法。

- 数据的聚合。
- 降低维度：聚类和分类。
- 标准化和归一化。



有些方法属于工程方法，有些方法属于人工智能或机器学习的范畴。

2.4 复杂业务模型下的故障定位

业务模型（或系统部署结构）复杂带来的最直接影响就是定位故障很困难，发现根源问题成本较高，需要多部门合作，开发、运维人员相互配合分析（现在的大规模系统很难找到一个能掌控全局的人），即使这样有时得出的结论也不见得各方都认可。

在开发层面，应对复杂业务的一般思路是采用 SOA、微服务化等，但从运维的角度讲，完成微服务化并没有降低业务的复杂度（当然结构肯定变清晰了）。

在这里，又不得不强调工程能力的重要性。在复杂、异构和各种技术栈混杂的业务系统中，如果想定位故障和发现问题，在各个系统中就必须有一个可追踪、共性的东西。然而，在现实中若想用某个“体系”来一统天下，则基本不可能，因为各种非技术因素可能会让这种努力一直停留在规划阶段，尤其是大公司，部门之间的鸿沟是技术人员无法跨越的。

所以，下面给出的几种简单方法和技术，既能在异构系统中建立某种关联，为智能化提供一定的支持，又不要求开发人员改变技术栈或开发框架。

- 日志标准化：日志包含所约定的内容、格式，能标识自己的业务线、服务层级等。
- 全链路追踪：TraceID 或者 RequestID 应该能从发起方透传到后端，标识唯一请求。
- SLA 规范化：采用统一的 SLA 约定，比如都用“响应时间”来约定性能指标，用“慢速比”来衡量系统健康度。

当这些工程（自动化、标准化）的水平达到一定高度后，我们才有望向智能化方向发展。

故障定位又称为告警关联（Alarm Correlation）、问题确定（Problem Determination）或根源故障分析（Root Cause Analysis），是指通过分析观测到的征兆（Symptom），找出产生这些征兆的真正原因。^[2]

在实践中通常用于故障定位的机器学习算法有关联规则和决策树。

还有很多方法，但笔者也在探索中，所以无法推荐一个“最佳”方法。究竟什么算法更适合，只能取决于实践中的效果了。

需要注意的是，并不是用了人工智能或机器学习，故障定位的效果就一定很好，这取决于



很多因素，比如特征工程、算法模型、参数调整、数据清洗等，需要不断地调整和学习。还是这句话：智能化的效果不仅仅取决于算法，工程能力也很重要，而且好的数据胜过好的算法。

2.5 本章小结

本章介绍了智能运维的定义和发展现状，智能运维需要解决的问题有：海量数据存储、分析、处理，多维度，多数据源，信息过载，复杂业务模型下的故障定位。在每一类问题后面，都给出了经过实践证明的解决方案和思路，同时也说明了为什么要这么做，以及在工程和算法上会遇到的问题。

2.6 参考文献

- [1] Gartner.<https://blogs.gartner.com/andrew-lerner/2017/08/09/aiops-platforms/>
- [2] 张成，廖建新，朱晓民. 基于贝叶斯疑似度的启发式故障定位算法



第 2 篇

站在巨人肩上：智能运维基础设施

“不积跬步，无以至千里；不积小流，无以成江海。”荀子在《劝学篇》里讲到，任何学问都需要积累。运维技术也在不断创新积累，在这个过程中经历了从工具到平台再到智能的巨大变革，所涉及的技术也非常多，而且很多技术已经发展成熟，尤其是当开源项目逐渐被大规模应用到各个企业中时，运维技术也进入了高速的发展阶段。在众多优秀的开源框架基础上，运维开发工程师更多的是要学习和掌握这些技术及工具，站在巨人肩上，利用这些优秀的基础设施，更多地关注业务，而不需要从零开始开发类似的工具和框架。

比如，在大数据存储和分析处理方面，Google 的“三驾马车”（Google 公开发布了关于 GFS、MapReduce 和 BigTable 的三篇论文），以及其在开源界对应的 HDFS、Hadoop 和 HBase，对互联网企业产生了极大的影响。随着数据规模的扩大，以及对性能的更高要求，后来 Google 发布了 Caffeine、Dremel 和 Pregel，它们被视为新时期的“三驾马车”，用于处理更大规模的数据。Caffeine 是 Google 出于自身需求的设计，使 Google 能够更迅速地将新的链接添加到大规模的网站索引系统中。Dremel 是一种分析信息的方式，Dremel 可跨越数千台服务器运行，允许“查询”大量的数据。Dremel 设法将海量的数据分析与对数据的深入挖掘进行有机的结合，可在大约 3 秒的时间里处理 1PB 的数据查询请求。Apache 后来开源的 Drill 就是 Dremel 的开源版本。Pregel 则是大规模分布式图计算平台，专门用来解决网页链接分析、社交数据挖掘等实际应用中涉及的大规模分布式图计算问题。

又如，在 AI 时代，有非常多的开源机器学习平台，如 Google 的 TensorFlow、百度的 PaddlePaddle 等。TensorFlow 是一个采用数据流图（Data Flow Graph），用于数值计算的开源软件库。在图中节点（Node）表示数学操作，线（Edge）则表示在节点间相互联系的多维数据数组，即张量（Tensor）。其灵活的架构让你可以在多种平台上展开计算，例如台式计算机中的一个或多个 CPU（或 GPU）、服务器、移动设备等。TensorFlow 最初是由 Google 大脑小组（隶属于 Google 机器智能研究机构）的研究员和工程师开发出来的，用于机器学习和深度神经网络



方面的研究，但这个系统的通用性使其也可广泛用于其他计算领域。在国内，百度在机器学习、深度学习技术方面比较领先，百度也开源了自己的深度学习平台 PaddlePaddle，它是并行分布式全功能深度学习框架，支持海量图像识别分类、机器翻译和自动驾驶等多个领域的业务需求。类似的开源工具和平台也被逐渐集成在云平台中作为大型云服务企业的基础服务，逐渐被工程化，同时极大地降低了开发工程师的学习门槛。

工欲善其事，必先利其器。第 2 篇我们将重点系统介绍如何使用开源框架进行大数据存储、处理、分析和计算，并结合一些应用案例逐渐展开。本篇主要分为以下几个章节：

- 第 3 章 开源数据采集技术
- 第 4 章 分布式消息队列
- 第 5 章 大数据存储技术
- 第 6 章 大规模数据离线计算分析
- 第 7 章 实时计算框架
- 第 8 章 时序数据分析框架
- 第 9 章 机器学习框架



第 3 章

开源数据采集技术

对业务指标的监控本质上是对数据的监控，所以说智能运维是建立在数据基础之上的。我们经常接触到的日志数据有 Web Server 的日志（如 Nginx 或者 Apache 的 access 和 error 日志）和 Linux 的系统日志（一般在/var/log 路径下），而业务数据可能是 JSON 格式的，也可能是 HTML，甚至是二进制文件，还可能是图片、视频或者音频文件。类似这样的结构化和非结构化数据，为数据采集和分析增加了极大的难度。值得庆幸的是，开源界出现了大量用于数据采集和预处理的工具与框架，开发者极易根据自己的业务需求和不同场景来选择或者进行二次开发。

3.1 数据采集工具对比

数据是监控报警的基石，因此我们在实现海量数据的分析监控前，就需要有一个顺手的工具来收集这些数据。现在市面上的日志收集工具琳琅满目，常见的有 Flume、Filebeat、Logstash、Scribe 等，而从中选择一个高可靠、高性能、占用系统资源少，以及符合业务应用场景等特性的工具就成了我们的首要工作。如表 3-1 所示，对比了几个开源日志收集工具。

表 3-1 开源日志收集工具对比

	Flume	Filebeat	Logstash	Scribe
语言	Java	Go	Ruby	C++
占用系统资源	一般	少	多	一般
扩展性	好	好	好	差
日志过滤	支持	支持	支持	不支持
日志解析	不支持	不支持	支持	不支持

通过对比可以发现，Logstash 虽然功能更加强大，但是占用系统资源较多，而 Filebeat 则更



加轻量级，占用系统资源较少。因此，本章我们选择 Filebeat 作为日志收集工具进行详细介绍。

3.2 轻量级采集工具Filebeat

Filebeat^[1]是 Elastic 开源 Beats 系列采集器中的轻量级日志采集器。当我们面对海量的日志需要采集，同时又不想占用太多的机器资源，以避免影响线上服务的运行时，Filebeat 就是一个不错的选择。

Filebeat 监控客户端的指定目录或者指定文件、跟踪文件，并将它们转发到 Kafka 或者直接发送到 Elasticsearch 中

3.2.1 Filebeat工作原理

Filebeat 启动一个或者多个 Prospector 去配置文件中指定日志路径下的勘查文件，对于勘查到的每个文件都会启动一个 Harvester。每个 Harvester 都读取一个文件，并把文件中的数据发送到 Libbeat，Libbeat 会聚合接收到的事件，并把聚合后的数据发送给配置文件中指定的 Output，如图 3-1 所示。

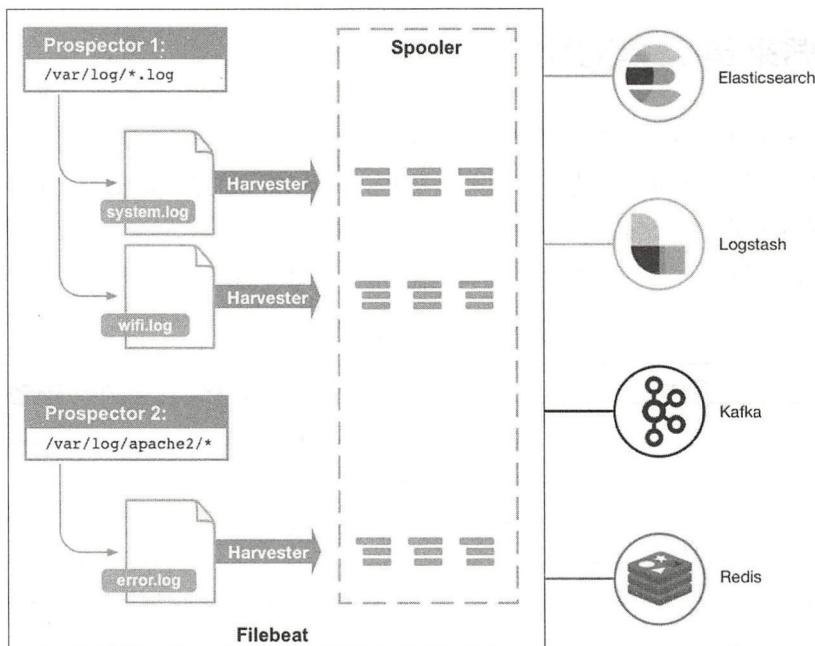


图3-1 Filebeat收集数据流程图



在上面的描述中可以发现，Filebeat 有两个重要的组件：Prospector 和 Harvester。那么 Prospector 和 Harvester 具体是怎么一起工作，将数据收集起来并发送出去的呢？

1. Prospector

Prospector 负责管理 Harvester，并发现所有可读的数据。如果输入的文件类型是 log，那么 Prospector 在磁盘上找出所有匹配指定全局路径的所有文件，并为每个文件启动一个 Harvester。下面的示例分别展示了从一个指定目录匹配文件收集数据和从一个指定文件收集数据。

```
filebeat.prospectors:
- type: log
  paths:
    - "/var/log/origin-*"
    - "/var/log/error.log"
```

Filebeat 当前支持两种勘查类型：log 和 stdin。在同一个配置文件中，每种勘查类型都能够被定义多次。log 勘查会检查文件以确定是否需要启动一个 Harvester 来收集数据、是否有一个 Harvester 已经在运行，或者这个文件是否被忽略。如果一个文件之前已经通过 Harvester 收集过数据，并且已经关闭，那么当这个文件大小发生改变后，新的 Harvester 将被启动，并且只会获取新增加的数据，不会再去收集已经收集过的数据。

2. Harvester

Harvester 负责打开文件，开始逐行读取单个文件的内容，并将读取到的数据发送到 Output。如果文件在经过 close_inactive 参数指定的时间后还没有更新，Harvester 将关闭，同时释放文件描述符，Harvester 会在经过 scan_frequency 指定的时间周期后重新启动收集数据。close_inactive 默认设置为 5 分钟，也就是说，Harvester 会释放在 5 分钟内没有更新的文件。在配置这个参数时不建议设置太长的时间，如果时间设置得太长，Harvester 收集的文件描述符将一直不能得到释放，磁盘的可用空间会越来越来少。这是因为 Harvester 占用文件描述符，文件没有被真正删除所导致的。

3. Filebeat如何保持文件状态

Filebeat 通过固定周期将文件状态存储在磁盘的 Registry 文件中来记录每个文件的状态。这个状态就是 Harvester 读取的文件内容，并确保所有内容都被发送出去时记录的是最后一行的偏移量，如果 Output 如 Elasticsearch 或者 Kafka 等变得不可用时，Filebeat 将跟踪最后一次发送的状态，直到 Output 恢复可用时才会继续读取文件。文件状态信息被每个 Prospector 保存在内容中，当出现异常导致 Filebeat 退出或者需要重新启动 Filebeat 时，文件状态信息将从 Registry 文件中读取到内存中，Harvester 就知道从哪里开始收集文件中的内容了。



每个 Prospector 都会记录它扫描到的每个文件的收集状态信息，因为一个文件会被移动到其他地方，或者被重命名，所以通过文件名和路径不能辨认一个文件，Filebeat 对每个文件都会通过一个唯一标识来识别其是否已经被 Harvester 收集过。

如果每天都有大量的新文件被创建，那么 Registry 文件很快就会变得很大，在 Filebeat 中，我们可以通过 `clean_removed` 和 `clean_inactive` 这两个参数来控制 Registry 文件的大小。

4. Filebeat如何确保数据不丢失

Filebeat 通过将每次发送的数据状态都存储在 Registry 文件中来确保数据不丢失。如果发送的 Output 端没有返回确认信息，Filebeat 将会继续尝试发送上一次的数据，直到 Output 端返回给 Filebeat 确认接收信息为止。

当向 Output 端发送数据，或者还没有接收到 Output 端返回的确认信息时，如 Filebeat 因异常退出而关闭，那么在 Filebeat 重启后，将会把上一次发送的数据再发送一遍，以确保数据至少被接收一次。所以在 Output 端可能会出现重复的数据。我们可以通过 `shutdown_timeout` 来设置 Filebeat 关闭前等待的时间。

当使用 Kafka 作为 Output 时，如果不要求数据的完整性，则可以容忍少量的丢失数据。我们也可以通过设置 `required_acks` 参数来提高 Filebeat 发送数据的效率，`required_acks` 默认值为 1，表示等待 Kafka 接收副本返回确认信息；设置为 0，表示 Kafka 不返回确认接收信息，Filebeat 将会持续发送；设置为 -1，表示需要等待 Kafka 所有副本确认接收信息后，才继续发送。

5. 性能特性

- 稳定可靠：Filebeat 会记录每次读取日志的 offset 值，如果出现异常导致进程中断，那么恢复后，Filebeat 可以从中断前的位置继续读取，从而保证数据不会丢失。
- 自动流控：当 Filebeat 向 Kafka 或者 Elasticsearch 等接收端写入数据时，如果接收端处理数据缓慢，Filebeat 将自动减缓读取日志的速度，以免造成日志拥堵。当接收端恢复正常后，Filebeat 将继续读取日志并发送给 Kafka 或者 Elasticsearch 等接收端。

3.2.2 Filebeat的安装与配置

1. 配置一般选项

`registry_file`：存储文件状态的文件，如果给出的是相对路径，那么文件所在的目录在 `${path.data}` 中指定。



shutdown_timeout: Filebeat 关闭前等待的时间, 这个时间用来让已经在发送的数据完成发送和接收端回复确认信息。

tags: 可以设置多个标签, 用来区分不同的服务, 如 `tags: ["web", "nginx", "ad"]`。

fields: 可以在 `fields` 字段中添加自定义的信息随日志一起输出。

2. 配置Prospector

Filebeat 使用 **Prospector** 来定位和处理文件, 我们可以在 Filebeat 的配置文件中指定一个或多个 **Prospector**。每个 **Prospector** 配置都以 “-” 符号开始, 在 “-” 的后面是一个 **Prospector** 的配置信息, 如文件路径、扫描周期等参数信息, 具体如下。

type: 输入数据的类型, 默认为 `log`, 还可以是 `stdin`、`redis`、`udp`、`docker` 等类型。

paths: 需要收集的日志文件绝对路径列表, 每行一个路径, 以 “-” 符号开始, 支持通配符, 如 “`/var/log/*.log`”, 表示匹配目录 “`/var/log/`” 下所有以 `.log` 结尾的日志文件。可以通过设置 `recursive_glob` 参数来递归查找指定目录下所有子目录中的日志文件。

exclude_lines: Filebeat 将会过滤所有匹配正则表达式的行。例如, 下面的例子会丢弃所有以 `DEBUG` 开始的行。

```
filebeat.prospectors:
- type: log
  paths:
    - "/var/log/nginx.log"
  exclude_lines: ['^DEBUG']
```

include_lines: Filebeat 只收集匹配正则表达式的行。例如, 下面的例子只收集以 `ERR` 或者 `WARN` 开始的行。

```
filebeat.prospectors:
- type: log
  paths:
    - "/var/log/nginx.log"
  include_lines: ['^ERR', '^WARN']
```

exclude_files: Filebeat 将会忽略收集匹配正则表达式的文件。

```
filebeat.prospectors:
- type: log
  paths:
```



```
- "/var/log/nginx.log"
exclude_files: ['\.gz$']
```

tags: 为 Filebeat 收集到的每条数据添加标签，标签可以有一个或者多个。当多种日志存在于 Kafka 的同一个 Topic 中时，可以通过 tags 来区分过滤指定的日志。

fields: 在输出信息中可以通过 fields 字段添加额外的信息。比如，可以添加一个用来区分日志的字段，字段可以是字符串、数组、字典或者任何嵌套的组合。在默认情况下，在 fields 中添加的字段将以字典的形式出现在输出信息的 fields 字段中。要将自定义字段存储为顶级字段，请将 fields_under_root 参数设置为 true。如果在一般配置中声明了一个重复字段，那么它的值将被这里声明的值覆盖。

```
filebeat.prospectors:
- type: log
  paths:
    - "/var/log/nginx.log"
  fields:
    user_id: admin
```

ignore_older: 过滤在指定时间前被修改的文件。对于日志长时间保存的情况，我们可以通过这个参数来忽略一段时间以前的文件。该参数默认设置为 0，表示不开启。我们可以通过“2h”或者“5m”等形式来指定时间。同时这个参数会被下面两种策略所影响：

- 文件从来没有被收集。
- 文件被收集，但是文件未更新的时间已经超过 ignore_older 指定的时间。

对于之前从未出现的文件，偏移量将被设置在文件的末尾；如果一个状态已经存在，那么偏移量不会改变；如果稍后文件被更新了，那么 Harvester 将从偏移量的位置继续读取。

ignore_older 依赖文件的修改时间，以确定文件是否被忽略。如果要从 Registry 文件中删除文件之前收集的状态信息，则可以使用 clean_inactive 参数。

在一个文件被 Prospector 忽略之前，它必须被关闭。为了确保一个被忽略的文件不会再被收集，我们必须设置 ignore_older 的时间为大于 clean_inactive 的时间。

如果当前正在收集的文件已经处于 ignore_older 状态，那么 Harvester 会先完成文件的收集，并且在 close_inactive 指定时间到达后关闭它，然后这个文件将会被忽略。

close_inactive: 如果一个文件在 close_inactive 指定时间内没有更新，那么 Filebeat 将关闭



文件句柄。这里的时间周期从文件的最后一行被 Harvester 读取开始计算，而不是基于文件的修改时间的。如果被关闭的文件发生更新，那么在经过 `scan_frequency` 指定的时间周期后，将启动新的 Harvester 读取更新的内容。我们可以将 `close_inactive` 的值设置为大于文件的最小更新频率。比如，如果文件几秒钟更新一次，则可以将 `close_inactive` 设置为 1 分钟。如果多个文件有不同的更新频率，则可以使用多个 Prospector 来设置不同的值。如果设置 `close_inactive` 的值比较小，那么文件句柄将很快被关闭。但是这样做也会导致文件更新的内容不能被实时收集。

close_renamed: 如果文件名被修改了，Filebeat 将关闭文件句柄。在默认情况下，Harvester 将一直打开并读取文件内容，因为文件句柄不依赖文件名。开启这个参数后，如果文件名被修改或者文件被移动到 Prospector 指定路径以外的地方，这个文件将不会被收集，Filebeat 也不会完成文件的收集。

close_removed: 如果一个文件被删除了，Filebeat 将关闭 Harvester。在正常情况下，一个文件只有在 `close_inactive` 指定的时间内没有更新才会被删除，但是如果一个文件被删除得比较早，并且我们没有设置 `close_removed`，那么 Filebeat 将保持文件打开状态以确保文件被收集完成。如果开启这个参数，文件可能不会被完整地收集，因为它们可能已经从磁盘上被删除了。该参数默认为开启状态。如果关闭该参数，还必须要关闭 `clean_removed`。

close_eof: 如果开启该参数，当 Filebeat 读取到文件末尾时，文件将很快被关闭。这对于文件只更新一次，而且不经常更新时很有用，比如当将每条日志写入一个单独的文件时。默认该参数处于关闭状态。

close_timeout: 如果开启该参数，Filebeat 将给予每个 Harvester 一个预定义的生存期，当 `close_timeout` 指定的时间过去后，无论读到文件的哪个位置，读取都将停止。这对于当文件只想用一段自定义的时间时非常有用，当自定义的时间过去后，`close_timeout` 将关闭这个文件。如果这个文件仍然在更新，Prospector 将在 `scan_frequency` 指定的时间周期后重新启动一个 Harvester 来收集数据，并且 `close_timeout` 对新的 Harvester 开始设置倒计时。这个参数在输出被阻塞时特别有用，可以使 Filebeat 持续打开文件句柄，即使是对于从磁盘上删除的文件也是如此。设置 `close_timeout` 为 5 分钟，这样可以使操作系统周期性地关闭并释放资源。该参数默认处于关闭状态。

clean_inactive: 当开启该参数后，Filebeat 将删除指定不活跃的时间周期后的文件状态，同时只有在文件被 Filebeat 忽略的情况下文件状态才能被删除。所以设置 `clean_inactive` 的时间必须大于 `ignore_older` 加上 `scan_frequency` 的时间，以确保处于被收集状态下的文件状态不会被删除；否则，因为 `clean_inactive` 被删除的文件就会被 Prospector 重新探测到，如果一个文件再



次出现或者更新，这个文件就会从头开始被 Harvester 读取，那么 Filebeat 就会重新发送这个文件的全部内容。

clean_inactive 的设置对于减小 Registry 文件的尺寸非常有用，特别是在每天产生大量文件的情况下。同时对于阻止 Linux 系统文件描述符的减少它也非常有用。

clean_removed: 当开启这个参数后，对于无法从磁盘上找到最后一个已知的名称的情况，Filebeat 将从 Registry 中清除该文件，这意味着被 Harvester 收集完成的文件被重命名后将被删除。该参数默认处于开启状态。

如果一个共享磁盘短暂地消失后又出现了，那么所有的文件都将从头开始重新读取，因为所有的文件状态都已经从 Registry 注册表中删除了。对于这样的情况，建议关闭 **clean_removed** 参数。

scan_frequency: 设置 Prospector 在指定路径下检查新文件的频率。如果需要接近实时地发送日志，则不要使用非常低的 **scan_frequency**，可以通过调整 **close_inactive** 让文件保持打开状态，并不断地对文件进行轮询。该参数默认值为 10 秒。

harvester_buffer_size: 每个 Harvester 获取一个文件时的缓冲区大小，默认值为 16384。

max_bytes: 单条日志可以拥有的最大字节数，在 **max_bytes** 之后的所有字节都将被丢失而不发送。这个参数对于多行设置尤其有用。其默认值为 10MB。

json: 用于解析 JSON 格式的日志消息。由于 Filebeat 是逐行处理日志的，所以 JSON 解析只有在每行一个 JSON 对象时有用。

keys_under_root: 默认解析出来的 JSON 信息在输出的 “json” 关键字下。如果开启这个参数，在输出文档中 Key 将被复制到顶层。该参数默认处于关闭状态。

overwrite_keys: 如果开启 **keys_under_root** 参数，那么 JSON 解析对象将覆盖 Filebeat 常用字段，以防止冲突。

add_error_key: 如果开启这个参数，Filebeat 将增加 **error.message** 和 **error.type**: json 两个字段，以防止 JSON 解析出错或者配置文件定义的 **message_key** 字段不能使用。

message_key: 可选参数，指定行过滤和多行设置的 JSON Key。如果指定的 Key 必须位于 JSON 对象的顶层，那么与 Key 关联的值必须是一个字符串；否则，过滤和多行聚合就不会发生。



tail_files: 如果开启该参数，那么 Filebeat 将从一个新文件的默认位置开始读取数据，而不是从文件的头部开始读取的。这个参数被应用在还没有被 Filebeat 处理过的新文件上，如果一个文件在开启这个参数之前已经被 Filebeat 处理了，且文件状态已经保存在注册表中，那么该参数将不会生效，Harvester 将继续从之前的位置开始读取数据。我们可以通过这个配置参数来过滤第一次收集的文件中旧的日志行。

symlinks: 该配置参数允许 Filebeat 获取符号链接文件。如果一个 Prospector 同时配置了符号链接文件和源文件，那么 Prospector 将处理它发现的第一个文件。但是如果两个不同的 Prospector 分别配置了符号链接文件和源文件，那么这两个文件都会被收集，Filebeat 将会发送两份数据到 Output，同时这两个 Prospector 将会相互覆盖对方的文件状态。

backoff: 设置 Filebeat 等待多长时间检查文件的更新，默认值为 1 秒，这意味着如果有新行产生，Filebeat 将每秒钟检查该文件一次，接近实时更新。每当文件中出现新行时，backoff 的值就会被重置。

harvester_limit: 设置同一个 Prospector 并行开启 Harvester 的最大数量，该设置也可以用来限制打开的文件句柄的数量。其默认值为 0，表示不做限制。

max_message_size: 当 Prospector 使用 UDP 类型时，指定通过 UDP 发送消息的最大字节数，默认值为 10240。

3. 配置Output

目前 Filebeat 的 Output 支持 Elasticsearch、Logstash、Kafka、Redis、File、Console 等。一般我们会将数据放入 Kafka 缓存，这样可以减小对后端的写入压力。

output.kafka:

```
hosts: [ "172.16.24.45:9110", "172.16.24.46:9110", "172.16.24.23:9110",
"172.16.24.38:9110", "172.16.24.50:9110" ]
version: 0.10.0.0
topic: "Topic"
username: "admin"
password: "xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
partition.round_robin:
  reachable_only: false
codec.format:
  string: "%{[message]}"
client_id: "log"
compression: "gzip"
```



```
retry.backoff: 1000
bulk_max_size: 100000
channel_buffer_size: 5000
required_acks: 1
```

hosts: Kafka Broker 地址，用来获取 Kafka 集群元信息。

version: 设置 Kafka 版本信息，默认值为 0.8.2.0 版本。如果想对每条数据增加写入 Kafka 的时间戳，则需要将 version 设置成 0.10.0.0 以上版本。

topic: 用来发送数据的 Kafka Topic 名称。

username: 如果 Kafka 设置了认证功能，则需要提供 username 来访问。

password: 如果 Kafka 设置了认证功能，则需要提供 password 来访问。

partition: 数据写入 Kafka 的每个分区的策略，默认为 hash。

- random: 随机发送数据到 Kafka 的分区中。
- round_robin: 轮询发送数据到 Kafka 的每个分区中。
- hash: 通过对指定字段做 hash 发送到对应的分区中。

reachable_only: 默认所有的分区都会接收数据，当一个 Partition Leader 不可用时，Output 可能会变成不可用状态，这时可以通过设置 reachable_only 为 true 来改变这种行为，将数据只发送给可用的分区。

codec.format: 发送数据的格式，默认以 JSON 格式发送。如果日志不是 JSON 格式的，则可以设置成按原数据格式发送。

client_id: 设置 client_id 用于日志、调试或者认证等，默认值为 beats。

compression: 设置数据输出压缩格式，默认值为 gzip，还支持 Snappy、LZ4 等。

retry.backoff: 在 Kafka Leader 选举期间重试的等待时间。

bulk_max_size: 在单次 Kafka 请求中，批量发送的最大事件数量，默认值为 2048。可以调大该值，提高单次发送的事件数量，但是不要超过 Kafka 中设置的接收最大值。

channel_buffer_size: 缓存在 Output 管道中的每个 Kafka Broker 消息数量。



required_acks: 设置是否需要等待 Kafka 返回数据接收确认信息，默认值为 1，表示需要等待接收的副本返回确认信息；设置为 0，表示 Kafka 不返回确认信息，Filebeat 持续发送；设置为-1，表示需要等待 Kafka 所有副本返回确认信息。对于对数据完整性要求不高的情况，则可以设置为 0 来提高数据发送速度。

4. 配置过滤和添加数据信息

Filebeat 提供了数据过滤和增加数据信息的功能，通过该功能我们可以过滤掉日志中不想要的信息，也可以在原来的日志中添加一些自己想要的内容。

Filebeat 在每个 Prospector 中都设置了 `include_lines`、`exclude_lines` 或者 `exclude_files` 参数，来提取或者过滤指定的内容或者文件。同时，这些参数支持正则匹配，可以在不同的 Prospector 中指定不同的过滤内容。通过通配符（*）提取 Nginx 日志文件的范例如下：

```
paths:
  - "/var/log/nginx.log.*"

  include_lines: ['nginx']
```

5. 配置日志

它出现在配置文件中的 `logging` 处，是用来配置 Filebeat 服务的日志输出的。如果没有明确指定日志输出位置，日志系统将把 Filebeat 的输出日志写到 `syslog` 或者 `rotate log` 中。

```
logging.level: info
logging.to_files: true
logging.to_syslog: false
logging.files:
  path: /var/logs/
  name: filebeat.log
  rotateeverybytes: 52428800
  keepfiles: 2
```

logging.level: 定义日志输出级别，有 `info`、`debug`、`warning` 和 `error` 级别。

logging.to_files: 设置日志输出到文件。

logging.to_syslog: 设置为 `true`，日志将被写入 `syslog` 中。

logging.files.path: 日志输出目录。

logging.files.name: 日志输出名称。



logging.files.rotateeverybytes: 一个日志文件的最大尺寸，当达到这个值后，Filebeat 日志将会自动轮转写到一个新的日志文件中。其默认值为 10MB。

logging.files.keepfiles: 在磁盘上保存的最近轮转的日志文件数量，当超过该值时，最老的日志文件将被删除。其默认值为 7。

6. 配置路径

可以通过在配置文件中设置 **path** 选项来指定配置文件、日志等信息的路径。

```
path.home: /data0/monitor/filebeat
path.config: ${path.home}
path.data: ${path.home}/data
path.logs: ${path.home}/logs
```

home: Filebeat 安装的家目录，这是其他路径设置的默认基本路径。

config: Filebeat YAML 配置文件的路径和 Elasticsearch 模板的路径，默认使用 home 目录。

data: Filebeat 数据文件存储路径，默认设置为 home 目录下的 data 子目录。

logs: Filebeat 日志文件路径，默认设置为 home 目录下的 logs 子目录。

7. 环境变量

可以在配置文件中使用环境变量来设置其他部分需要引用的值。例如，当 **path.home** 定义后，如果在配置文件中用到了同样的配置，则可以用环境变量来引用其定义的内容，赋值给其他配置参数。下面通过 **\${path.home}** 让 **path.config** 调用了 **path.home** 定义的内容。

```
path.home: /data0/monitor/filebeat
path.config: ${path.home}
```

8. 配置示例

```
#===== Filebeat global options =====
filebeat:
  registry_file: ".nginx"
  shutdown_timeout: "5s"

#===== Filebeat prospectors =====
filebeat.prospectors:
-
  close_inactive: "1h"
  scan_frequency: "10s"
```




```

ignore_older: "6h"
clean_inactive: "8h"
tail_files: true
harvester_buffer_size: 10485760
backoff: "1s"
paths:
  - "/var/log/nginx.log.*"
include_lines: ['nginx']
exclude_lines: ['DEBUG']

#===== Outputs =====
#----- Kafka output -----
output.kafka:
  hosts: [ "172.16.24.45:9110", "172.16.24.46:9110", "172.16.24.23:9110",
"172.16.24.38:9110", "172.16.24.50:9110" ]
  version: 0.10.0.0
  topic: "nginx"
  username: "admin"
  password: "xxxxxxxxxxxxxxxxxxxxxx"
  partition.round_robin:
    reachable_only: false
  codec.format:
    string: "%{[message]}"
  client_id: "nginx"
  compression: "gzip"
  retry.backoff: 1000
  bulk_max_size: 100000
  channel_buffer_size: 5000
  required_acks: 1

#===== Paths =====
path.home: /data0/monitor/filebeat
path.config: ${path.home}
path.data: ${path.home}/data
path.logs: ${path.home}/logs

#===== Logging =====
logging.level: info
logging.to_files: true
logging.to_syslog: false

```



```
logging.files:
  path: /data0/monitor/filebeat/logs/
  name: nginx.log
  rotateeverybytes: 52428800
  keepfiles: 2
```

3.2.3 启动和运行Filebeat

通过 `supervisord` 来管理进程，配置 `supervisord` 启动 `Filebeat` 的配置文件。

```
[program:nginx]
command=/data0/monitor/filebeat/filebeat -c /data0/monitor/filebeat/nginx.yml
numprocs=1
autostart=true
startretries=3
autorestart=true
user=root
redirect_stderr=true
stdout_logfile=None
```

执行下列命令启动 `Filebeat`：

```
supervisorctl start nginx
```

3.3 日志采集解析工具

`Filebeat` 是轻量级的日志采集工具，它可以帮助我们很快地将所采集的日志发送到 `Kafka`、`Elasticsearch` 等输出端，但是 `Filebeat` 并不支持对采集到的日志进行解析；而 `Logstash`^[2] 就可以完成复杂的日志解析、数据提取等工作，同时相对于 `Filebeat`，`Logstash` 支持更多的输入输出方式。

- 输入：`Logstash` 支持各种数据输入，可以同时从 `Web` 应用、日志、指标、数据存储等数据源采集数据，再以连续的流式传输方式发送到各个过滤器和输出端。
- 过滤器：`Logstash` 过滤器能够解析从输入端来的各个事件，识别已命名的字段来构建结构，并将它们转换为通用格式，以便我们轻松地分析和实现商业价值。`Logstash` 还能够转换和提取数据，不受原来的格式和复杂度影响。目前 `Logstash` 的过滤器库支持 40 多种过滤方式，功能非常丰富。



- 输出：目前 Logstash 已提供 50 多种输出方式供选择，我们可以灵活地把指标数据发送到指定的地方，供下游使用。

3.3.1 Logstash工作原理

Logstash 事件处理流如图 3-2 所示。



图3-2 Logstash事件处理流

Input 产生数据；Filter 清洗数据，提取和转换我们需要的信息；Output 将所提取的数据发送到我们需要的地方。

1. Input

我们从 Input 端获取数据，常用的 Input 类型如下。

- file: 从一个文件中读取数据，就像使用“tail -f”命令一样。
- kafka: 以 Consumer 的方式从 Kafka 消费数据。
- filebeat: 由 Filebeat 推送数据给 Logstash。
- tcp: 开启一个监听端口，所有向这个端口发送的数据都可以被接收。

2. Filter

Filter 是 Logstash 流处理的中间环节，我们可以将多个 Filter 组合在一起使用。常用的过滤器如下。

- grok: grok 用于非结构化数据解析，通过正则表达式任意构造自己想要的数据。不过 grok 一直被人诟病的是它的执行效率，毕竟通过正则表达式构造数据特别耗费系统资源。
- drop: 对于不匹配的数据直接丢弃。
- mutate: 对数据进行转换，可以进行重命名、删除、替换、修改字段等操作。
- split: 对目标字段以指定的分隔符分隔。

3. Output

Output 是 Logstash 事件流的最终环节。一个事件可以同时输出到多个 Output，常用的 Output 如下。



- `elasticsearch`: 将事件数据发送到 Elasticsearch。
- `kafka`: 将事件数据生产到 Kafka。
- `file`: 将事件数据写入本地指定文件中。
- `codecs`: `codec` 作为 Input 和 Output 的一部分, 可以对数据执行序列化操作。常用的 `codec` 包括 `son` (以 JSON 格式编码或解码数据) 和 `ultiline` (将多个事件合并到一个事件中)。

3.3.2 安装Logstash

Logstash 依赖 Java 8 及以上版本的支持, 所以在启动 Logstash 之前, 需要先配置 Java 环境。安装 Logstash 一般有以下三种方式。

1. 通过二进制文件安装

从官网将二进制形式的压缩包下载到本地磁盘, 直接解压缩就可以使用。

2. 通过RPM包安装

安装官方 YUM 源, 然后使用 `yum` 命令安装。

- 下载并安装公钥签名:

```
rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch
```

- 在 `/etc/yum.repos.d/` 目录下新增以 `.repo` 为后缀的文件。

```
[logstash-6.x]
name=Elastic repository for 6.x packages
baseurl=https://artifacts.elastic.co/packages/6.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
```

- 使用 `yum` 命令安装 Logstash。

```
sudo yum install logstash
```

3. 通过Docker安装

从 Docker 官网拉取 Logstash 的镜像。

```
docker pull docker.elastic.co/logstash/logstash:6.0.1
```



3.3.3 配置Logstash

Logstash 有两种类型的配置文件：管道配置文件，用来定义 Logstash 的事件处理流；环境配置文件，用来控制 Logstash 的启动。

Logstash 安装成功后，默认包含三个配置文件，分别是 `logstash.yml`、`jvm.options` 和 `startup.options`。

1. 环境配置文件

我们可以通过在环境配置文件中设置参数来控制 Logstash 的执行。比如可以指定管道的设置、配置文件的位置、日志选项，以及其他一些设置。`logstash.yml` 中的大部分设置也可以通过命令行的形式在 Logstash 启动时执行。通过命令行设置的参数配置将覆盖在 `logstash.yml` 文件中定义的相同配置。

`logstash.yml` 通过 YAML 语言编写。例如，可以以分层的形式定义管道批量的大小和批处理的延迟。

```
pipeline:
```

```
  batch:
```

```
    size: 125
```

```
    delay: 5
```

path.data: Logstash 的数据文件以及它的插件用于持久化的目录。如果用 Logstash 启动多个进程服务于不同的事件流，则需要为每个事件流都指定单独的 `path.data` 路径。

pipeline.workers: 在执行 Filter 和 Output 阶段并行执行的管道数量，默认为主机 CPU 的内核数量。

pipeline.batch.size: 在执行 Filter 和 Output 之前单个工作线程从 Input 中收集的最大事件数。设置一个比较大的批处理量通常能带来更好的执行效率，但是这样做也意味着更大的内存开销，所以设置不要超过 JVM 的大小，或者增加 JVM 堆的大小。

pipeline.batch.delay: 在将一组没有达到所设置的批量大小的事件发送到管道处理之前，需要等待的时间（单位是毫秒）。

pipeline.unsafe_shutdown: 在默认情况下，Logstash 在将内存中的数据全部发送出去之前拒绝退出。如果设置该参数为 `true`，那么 Logstash 将不会等待内存中的数据全部发送完就退出，这样会造成数据丢失。



`path.config`: 指定 Logstash 处理事件流的配置文件路径。

`config.reload.automatic`: 当处理事件流的配置文件内容发生改变时，自动重新加载配置文件。

`config.reload.interval`: Logstash 检查配置文件变化的时间周期，默认为 3 秒。

`queue.type`: 指定缓存时间的队列类型，可以选择内存队列类型 `memory` 或者磁盘持久化队列类型 `persisted`，默认为内存队列类型。当我们追求数据完整性，能够容忍处理效率的下降时，可以切换到磁盘持久化队列类型。

`path.queue`: 当 `queue.type` 为磁盘持久化类型时，设置数据文件存储在磁盘上的路径。

`dead_letter_queue.enable`: 开启 DLQ 特性。当 Logstash 遇到一个因为数据包含映射错误或其他问题无法处理的事件时，Logstash 管道要么挂起，要么丢弃该事件。为了防止在这种情况下数据丢失，我们可以配置 Logstash 将不成功的事件写入一个死信队列，而不是删除它们。每个写入死信队列中的事件，都包含原始事件、描述事件不能被处理的原因的元数据，编写事件的插件的信息，以及事件进入死信队列时的时间戳。要处理死信队列中的事件，需要创建一个 Logstash 管道，然后从 `dead_letter_queue` 队列中读取事件再写入 Output。

`dead_letter_queue.max_bytes`: `dead_letter_queue` 队列的最大字节数，超出这个值的事件将被丢弃。

`path.dead_letter_queue`: `dead_letter_queue` 队列的数据文件存储目录位置。

`http.host`: REST 指标绑定的地址。

`http.port`: REST 指标绑定的端口。

`log.level`: 设置日志级别。

`log.format`: 设置日志格式。

`path.logs`: 设置 Logstash 写日志的路径。

`path.plugins`: 定义 Logstash 插件安装目录。

2. Input 插件配置

目前 Logstash 的 Input 插件有近 50 种，在这里以最常用的 Kafka 来配置。



最新的 Kafka 插件使用的是 Kafka Client 1.0.0 版本。Logstash Kafka 消费管理使用默认的 offset 管理策略。在默认情况下，Logstash 实例通过一个逻辑组来订阅 Kafka Topic 中的消息。每个 Logstash Kafka Consumer 都能够运行多个线程来增加消费能力；否则，就需要跨物理机运行多个拥有相同 group_id 的 Logstash 实例来消费。Kafka 中的消息将分发到具有相同 group_id 的所有 Logstash 实例上。在理想情况下，应该设置 Logstash 的线程数和 Kafka 的分区数一致。

metadata fields: Kafka Broker 的元信息，默认消费后会添加在 @metadata 字段中。

- `[@metadata][kafka][topic]`: 消费的 Kafka Topic 名称。
- `[@metadata][kafka][consumer_group]`: 消费的 Group 名称。
- `[@metadata][kafka][partition]`: 消费消息的分区信息。
- `[@metadata][kafka][offset]`: 消费消息的 offset 记录。
- `[@metadata][kafka][key]`: 消费消息的 Key。
- `[@metadata][kafka][timestamp]`: Kafka Broker 接收消息的时间戳。

auto_commit_interval_ms: 消费 offset 提交给 Kafka 的时间间隔，默认值为 5000 毫秒。

auto_offset_reset: 指定第一次消费 Kafka 中消息的位置，可以取以下值。

- `earlist`: 从 Kafka 中最早的 offset 开始消费。
- `latest`: 从 Kafka 中新近的 offset 开始消费。
- `none`: 如果消费者没有发现之前的 offset，就抛出异常。
- `anything else`: 向消费者抛出异常。

bootstrap_servers: 连接 Kafka 集群的 Broker List 地址，形式如 host1:端口, host2:端口, host3:端口。

client_id: 标识消费 Kafka 集群的消费者 id，用于通过逻辑分组跟踪消费状态。

connections_max_idle_ms: 在指定时间后关闭空闲的连接。

consumer_threads: 消费者消费 Kafka Topic 的连接数，该参数应该和 Kafka 分区的数量保持一致。



enable_auto_commit: 定期向 Kafka 提交消费的 offset。

groupid: 消费者组消费 Kafka 的唯一标识。消费者组是一个逻辑分组，可以由多台机器组成，那么 Kafka Topic 中的消息将被分发到具有相同标识的消费者中。

heartbeat_interval_ms: Kafka 用来确保消费者保持活跃状态的时间，或者新的消费者加入和离开做 rebalance 的时间。该值必须低于 session.timeout.ms。

jaas_path: JAAS 认证文件的路径，用于在消费者消费 Kafka 时指定认证方式、用户名和密码等。JAAS 文件如下：

```
KafkaClient {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  username="test"
  password="XXXXXXXXXXXXXXXXXXXXXXXXXXXX";
};
```

sasl_mechanism: 客户端连接的 SASL 机制，默认的机制是 GSSAPI。

security_protocol: 客户端连接的安全协议，如 PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL，默认为 PLAINTEXT。

session_timeout_ms: 如果一个消费者在没有调用 poll_timeout_ms 的时间到达后，则会标记死用户，Kafka 会为相同 group_id 标识的一组消费者触发 rebalance 操作。

topics: 消费的 Kafka Topic 列表

3. Filter 插件配置

目前 Logstash 支持 40 多种 Filter 插件，下面列举几个常用的插件。

(1) date 插件

date 插件用来从日志指定字段中解析时间，然后使用该时间或者时间戳作为 Logstash 事件的时间。date 插件对于时间排序和老的事件写入尤其重要，如果一个事件中没有记录 date 字段，那么之后查询时它们将不能按顺序排列。

match: 指定 date 字段，并指定匹配的时间格式。如下例所示，队列的第一个字段表示 date 字段匹配的时间字段为“time”，第二个字段表示 time 字段匹配的时间格式为“UNIX_MS”。

```
match => [ "time", "UNIX_MS" ]
```

如果多个事件拥有不同的时间格式，则可以在 match 的列表中指定多种时间匹配格式。



例如：

```
match => [ "time", "UNIX_MS", "MMM dd yyyy HH:mm:ss", "ISO8601" ]
```

常用的时间格式如下。

- ISO8601：如 2018-03-25T08:01:01.100Z。
- UNIX：秒 UNIX 时间戳，如 1521936061。
- UNIX_MS：毫秒 UNIX 时间戳，如 1521936061100。
- MMM dd yyyy HH:mm:ss：匹配如 Mar 25 2018 08:01:01 的格式。
- yyyy-MM-dd HH:mm:ss：匹配如 2018-03-25 08:01:01 的格式。

(2) dissect 插件

`dissect` 用于进行分割操作，将一组分隔符应用到字符串值。`dissect` 不使用正则表达式，并且解析非常快。但是如果日志的行与行之间的格式结构不一样，那么使用 `grok` 会更合适。

`dissect` 使用 “%{}” 来描述字段。比如下面的例子表示，匹配用空格分隔的 a、b、c 三个字段。

```
%{a} %{b} %{c}
```

- %{}：跳过该字段。
- %{foo}：匹配 foo 字段。
- %{?foo}：跳过一个命名的字段。
- %{+foo}：将该字段附加到前一个字段后。
- %{+foo/2}：指定附加到指定字段的位置。
- %{?foo} %{&foo}：将第二个字段的值赋给 foo。

`dissect` 使用举例：

```
dissect {
  mapping => {
    "msg" => "%{}: %{?msgTotal}:%{&msgTotal} %{?msgSuccess} %{status}"
  }
}
```



```
convert_datatype => { msgTotal => "int" }
}
```

(3) drop 插件

使用 drop 插件可以丢弃符合条件的事件，我们还可以通过 percentage 设置丢弃事件的百分比。例如，在下面的 Filter 中将丢弃 40% 的 loglevel 字段值等于 debug 的事件。

```
filter {
  if [loglevel] == "debug" {
    drop {
      percentage => 40
    }
  }
}
```

(4) grok 插件

grok 是一种将非结构化日志数据解析为结构化和可查询的内容的工具，通过自定义的文本格式来匹配日志。

grok 表达式的语法格式形如 %{SYNTAX:SEMANTIC}。其中 SYNTAX 为匹配字段的模型，如 3.14 匹配 NUMBER 模型，192.168.1.1 匹配 IP 模型；SEMANTIC 为匹配内容的名称，如 3.14 可能是事件的请求耗时，可以将它命名为 cost_time，192.168.1.1 可能是主机 IP 地址，就可以命名为 host。所以结合上面的例子，可以在 grok 中这样定义：

```
%{NUMBER:cost_time:float} %{IP:host}
```

可以为字段指定数据类型，比如在上面的表达式中，指定了 cost_time 为 float 类型。默认所有字段都为字符串类型。

有了这种语法和语义的概念，我们可以从一个示例日志中提取有用的字段。比如这个虚构的 HTTP 请求日志：

```
192.168.1.1 GET /index.html 15824 0.043
```

那么 grok 表达式就可以像这样表示：

```
%{IP:client} %{WORD:method} %{URIPATHPARAM:request} %{NUMBER:bytes:int} %{NUMBER:cost_time:float}
```

grok 通过正则表达式来匹配文本，因此任何正则表达式语法在 grok 中都有效。

Logstash 默认提供大约 120 种文本模板，可以在 <https://github.com/logstash-plugins/logstash->



`patterns-core/tree/master/patterns` 中找到满足需求的模板。当然，也可以自定义匹配模板，然后在 <http://grokdebug.herokuapp.com> 或者 <http://grokconstructor.appspot.com/> 中测试该模板。

match: 匹配字段。

overwrite: 重写已存在的指定字段。

patterns: 在默认情况下，已经包含了 Logstash 定义的模板，如果有自定义的模板，则可以在这里指定将其包含进来。

patterns_files_glob: 匹配 `patterns_dir` 指定目录中包含的文件。

timeout_millis: 定义正则匹配超时时间，默认值为 30000ms。

(5) mutate 插件

mutate 允许对字段进行一些改变，比如重命名、移除、替换或者修改字段等。

convert: 改变字段数据类型，比如将一个字段从字符串类型变为整型。

copy: 将一个已存在的字段复制到另一个字段中，如果目标字段已存在，那么它将被覆盖。

gsub: 将指定字符串替换为所有正则匹配的内容，只支持字符串和字符串数组。

rename: 重命名指定字段。

replace: 替换指定字段的值，可以通过 “`%{foo}`” 将其他字段的内容赋值到该字段中。

split: 用指定分隔符分隔指定字段，只对字符串类型的字段有效。

update: 更新某个字段的值，如果这个字段不存在，则不执行任何操作。

以下配置选项为通用选项，支持所有的过滤器插件。

add_field: 增加一个自定义字段，字段内容可以通过 “`%{field}`” 来引用事件中的字段信息。

add_tag: 增加一个自定义标签，标签内容可以通过 “`%{tag}`” 来引用事件中的标签信息。

id: 对插件添加唯一 `id`，特别是对于一个配置文件中拥有多个相同的插件的情况，在监控 Logstash 时可以通过 `id` 来区分。



`remove_field`: 移除一个字段。

`remove_tag`: 移除一个标签。

4. Output 插件配置

目前 Output 插件已有 50 多种，常用的有 Elasticsearch、Kafka、File 等，这里以输出到 Elasticsearch 为例来配置 Output。

我们可以在 Output 中配置 Elasticsearch 输出，将日志写入 Elasticsearch 中。这里通过 HTTP 协议将数据传输到 Elasticsearch。

(1) 重试策略

Logstash 的 Elasticsearch 插件使用 Elasticsearch Bulk API 来优化写入的性能。如果返回的状态码不是 200，那么这个请求将被无限次重试。如果开启了 DQL，那么请求状态码为 400 和 404 的事件将被转发到 DQL 中；如果没有开启 DQL，将会发出一条日志消息，同时删除该事件。而返回状态码为 409 的事件将被删除。

(2) 批量大小

Elasticsearch 插件将多个事件批量发送为单个请求。但是如果请求大小超过 20MB，我们将在多个批处理请求之前将其分解；如果单个文档大小超过 20MB，那么它将作为单个请求发送。其参数解释如下。

`action`: 执行索引的动作，默认为 `index`。可用的动作如下。

- `index`: 索引一个文档。
- `delete`: 通过 `id` 删除一个文档。
- `create`: 创建一个文档。如果索引中已存在该文档 `id`，那么创建失败。
- `update`: 通过 `id` 更新一个文档。
- `document_id`: 指定索引的文档 `id`，如果已存在相同的 `id`，那么将覆盖已存在的文档。

`hosts`: 指定 Elasticsearch 实例地址。如果指定了一个主机列表，那么请求将被平均分发到多个实例上。建议不要写 Elasticsearch 集群的主节点。

`http_compression`: 开启请求 `gzip` 压缩，默认是关闭的。



index: 指定事件写入的索引名称，索引名称不能包含大写字母。如果需要按天建立索引，则可以像 `logstash-%{+YYYY.MM.dd}` 这样指定。

pool_max: 指定到 Elasticsearch 的最大连接数。

pool_max_pre_route: 指定到 Elasticsearch 每个节点的最大连接数。

routing: 按照指定字段路由分发请求。建议按照查询类型设置 **routing**，这有助于提高 Elasticsearch 的查询效率。

timeout: 设置请求超时时间，单位为秒。如果发生超时，请求将被重试。

codec: 指定输出数据的编解码器。

3.3.4 启动Logstash

通过 **supervisord** 来管理进程，配置 **supervisord** 启动 Logstash 的配置文件。

```
[program:nginx]
command=/data0/logstash/bin/logstash -f /data0/logstash/conf/nginx.conf --http.host
192.168.1.1 --http.port 9600 --pipeline.workers 2 --pipeline.batch.size 1000
--pipeline.batch.delay 60 --path.data /data0/logstash/data/nginx --path.logs
/data0/logstash/logs/nginx --config.reload.automatic
numprocs=1
autostart=true
startretries=3
autorestart=true
user=root
redirect_stderr=true
stdout_logfile=None
```

执行以下命令启动 Logstash。

```
supervisorctl start nginx
```

3.4 本章小结

日志收集工具一般和业务服务部署在一起，为了不和线上服务抢占资源，一定要选择占用资源少的日志收集工具，也不要再在收集端进行日志清洗和分析操作。我们应该将日志集中收集到一个地方，再统一进行清洗和分析。Filebeat 正适合部署在客户端来收集日志，而 Logstash



则更适合用在消息队列后进行统一的日志清洗和分析。

3.5 参考文献

[1] <https://www.elastic.co/guide/en/beats/filebeat/current/index.html>

[2] <https://www.elastic.co/guide/en/logstash/current/index.html>



第 4 章

分布式消息队列

在进行高并发系统架构设计时经常会使用到消息队列，它作为消息中间件，可以提高上游生产者写入的并发性能，提高系统吞吐量，方便下游消费者进行并行消费，提高读的并发性能。消息中间件是重要的消息传输交换的重要组件，传输的消息不同于 API 请求数据，消息生产者可以将数据存储在内存和磁盘上，这样可以等待消费者消费使用数据。消息中间件具有发布—订阅、请求—应答、管道三个特点。对消息中间件的应用可以让系统与系统之间的耦合度大大降低，同时让系统之间的业务处理异步化。分布式消息队列是系统实现最终一致性、可扩展性、高可用性的重要基石。

4.1 开源消息队列对比与分析

4.1.1 概述

目前市面上存在 ZeroMQ、ActiveMQ、RocketMQ、Kafka 等消息中间件，接下来就对这些消息中间件进行简要的说明。

4.1.2 ZeroMQ

ZeroMQ 是一种基于消息队列的多线程网络库，其对套接字类型、连接处理、帧甚至路由的底层细节进行抽象，提供跨越多种传输协议的套接字。ZeroMQ 是网络通信中新的一层，介于应用层和传输层之间（按照 TCP/IP 划分），它是一个可伸缩层，可并行运行，分散在分布式系统间。ZeroMQ 几乎所有的 I/O 操作都是异步的，主线程是不会被阻塞的。ZeroMQ 会根据用户调用 `zmq_init` 函数时传入的接口参数来创建对应数量的 I/O 线程。每个 I/O 线程都有与之绑定的 Poller，Poller 采用经典的 Reactor 模式实现，Poller 根据不同的操作系统平台使用不同的网



络 I/O 模型（如 select、poll、epoll、devpoll、kequeue 等）。主线程和 I/O 线程通过 Mail Box 传递消息来进行通信。当服务器开始监听或者客户发起连接时，在主线程中创建 `zmq_connector` 或 `zmq_listener`，通过 Mail Box 发消息的形式将其绑定到 I/O 线程，I/O 线程会把 `zmq_connector` 或 `zmq_listener` 添加到 Poller 中用以侦听读/写事件。当服务器与客户端第一次通信时，会创建 `zmq_init` 来发送标识，用以进行认证。当认证结束后，双方会为此次连接创建 Session，以后双方就通过 Session 进行通信。每个 Session 都会关联到相应的读/写管道，主线程收发消息是分别从管道中读/写数据的。Session 并不是实际地跟 Kernel 交换 I/O 数据，而是通过 Plugin 到 Session 中的 Engine 来与 Kernel 交换 I/O 数据的。ZeroMQ 开发者引入 JAR 包直接使用，不支持数据持久化，所以它适合在高吞吐量或低延迟的场景中使用，但是需要大量编码。

4.1.3 ActiveMQ

ActiveMQ 是一种开源的消息队列，实现了 JMS 1.1 规范的面向消息的中间件（MOM），为应用程序提供高效的、可扩展的、稳定的和安全的的企业级消息通信。ActiveMQ 的设计目的是提供标准的、面向消息的、能够跨越多语言和多系统的应用集成消息通信中间件。ActiveMQ 实现了 JMS 标准并提供了很多附加特性，包括 JMX 管理、主从管理（Master/Slave，这是集群模式的一种，主要体现在可靠性方面，当主代理出现故障时，从代理会替代主代理的位置，不至于使消息系统瘫痪）、消息组通信（同一组消息，仅会提交给一个客户进行处理）、有序消息管理（确保消息能够按照发送的次序被接收者接收）、消息优先级（优先级高的消息先被投递和处理）、订阅消息的延迟接收（订阅消息在发布时，如果订阅者没有开启连接，那么当订阅者开启连接时，消息中介将会向其提交之前的未处理的消息）、接收者处理过慢（可以使用动态负载均衡，将多数消息提交给处理快的接收者，这主要是针对 PTP 消息所说的）、虚拟接收者（降低与中介的连接数量）、成熟的消息持久化技术（部分消息需要持久化到数据库或文件系统中，当中介崩溃时，信息不会丢失）、支持游标操作（可以处理大消息）、支持消息的转换、通过使用 Apache 的 Camel 支持 EIP、使用镜像队列的形式轻松地对消息队列进行监控等。

4.1.4 RocketMQ

RocketMQ 是一种纯 Java 的、分布式的、队列模型的开源消息中间件，其前身是 MetaQ，当 MetaQ 3.0 发布时，产品名称改为 RocketMQ。RocketMQ 能够保证严格的消息顺序，提供丰富的消息拉取模式、高效的订阅者水平扩展能力、实时的消息订阅机制、亿级消息堆积能力。其具有高吞吐量、高可用性、适合大规模分布式系统应用的特点。目前 RocketMQ 在阿里巴巴集团被广泛应用于交易、充值、流计算、消息推送、日志流式处理、binglog 分发等场景，支撑



了阿里巴巴的多次“双 11”活动。

4.1.5 Kafka

Kafka^[1]是 LinkedIn 于 2010 年 12 月开发并开源的一个分布式流平台，现在是 Apache 的顶级项目，是一个高性能、跨语言、分布式、发布—订阅消息队列的系统，消费者通过拉取的方式消费消息。Kafka 消息中间件的特点是：快速持久化，可以在 $O(1)$ 的系统开销下进行消息持久化；高吞吐量，在一台普通的服务器上即可以达到 10W/s 的吞吐速率；完全的分布式系统，Broker、Producer、Consumer 都原生自动支持分布式，自动实现复杂均衡。因为 Kafka 在设计之初是作为日志流平台和运营消息管道平台的，所以实现了消息顺序和海量堆积能力。

4.2 Kafka的安装与使用

上述章节对消息队列做了简单的介绍和说明。由于 Kafka 具有高吞吐量、低延时特点，是完全的分布式系统，其使用开发成本较低并且可以保证消息顺序，因此我们采用 Kafka 作为微博广告消息中间件，它在微博广告监控、实时计算及结算业务中承载重要的作用。本节主要对 Kafka 的安装和使用进行介绍。

4.2.1 组件概念

Broker: 缓存代理，Kafka 集群中的一台或多台服务器统称为 Broker。

Topic: 区分 Kafka 的消息类型，将不同的消息写到不同的 Kafka Topic 中。

Partition: Topic 物理上的分组，一个 Topic 可以分为多个 Partition，每个 Partition 都是一个有序的队列。Partition 中的每条消息都会被分配一个有序 id (offset)。

Producer: 消息的生产者。

Consumer: 消息的消费者。

Consumer Group: 每一组消费者都有相同的 Group id，这样可以保证多个消费者不重复消费同一条消息。

4.2.2 基本特性

- 高吞吐量、低延迟：Kafka 每秒可以处理几十万条消息，它的延迟时间最低只有几毫



秒。每个 Topic 都可以分为多个 Partition，Consumer Group 对 Partition 进行消费操作。

- 可扩展性：Kafka 集群支持热扩展，在消费能力不足的情况下可以增加分区。
- 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份，防止数据丢失。
- 容错性：允许集群中节点失败，若副本数量为 n ，则允许 $n-1$ 个节点失败。
- 高并发性：由于 Kafka 是顺序读/写磁盘的，使用 NIO 的网络模型支持数千个客户端同时读/写。

4.2.3 安装与使用

1. 环境准备

- (1) 安装 Java，Kafka 依赖 Java 环境，本文使用 JDK 1.8.0_151。
- (2) 安装 Zookeeper，Kafka Broker 需要注册到 Zookeeper。
- (3) 安装 Kafka，直接从官网 (<https://kafka.apache.org>) 下载最新的 Release 版本。

2. 单机安装

- (1) 解压缩所下载的 Kafka 包。

```
tar -xzf kafka_2.11-1.0.1.tgz
```

- (2) 启动 Zookeeper。

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

- (3) 启动 Kafka Broker。

```
bin/kafka-server-start.sh config/server.properties
```

3. 集群安装

- (1) 复制 Kafka 的配置文件，修改 broker.id（如果是在同一台机器上部署，则修改监听端口号和日志的路径）。

```
cp config/server.properties config/server-1.properties
```

```
cp config/server.properties config/server-2.properties
```

- (2) 启动两个 Kafka Server（因为在同一台机器上），执行以下命令启动并创建 Topic，同时进行消息测试。



```
# 创建 Topic
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic test
# 向 Kafka 发送消息
./kafka-console-producer.sh --broker-list localhost:9092 --topic test
# 从 Kafka 消费消息
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test
--from-beginning
# 增加分区
./kafka-topics.sh --alter --topic topic1 --zookeeper zk1p:2181/kafka --partitions 6
```

4.2.4 Java API的使用

Kafka Java API 分为 High Level 和 Low Level 两种。High Level Consumer API 围绕着 Consumer Group 这个逻辑概念展开，它屏蔽了每个 Topic 中的每个 Partition 的 offset 管理（自动读取 Zookeeper 中该 Consumer Group 的 Last offset）、Broker 失败转移，以及增减 Partition、Consumer 时的负载均衡（当 Partition 和 Consumer 增减时，Kafka 自动进行负载均衡）。

Java 访问 Kafka 集群需要引入 JAR 包，我们通过 Maven 管理 JAR 包。在 Java 工程的 POM 文件中引入下面的 JAR。

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>1.0.1</version>
</dependency>
```

生产 Kafka 消息：

```
public class UserKafkaProducer extends Thread
{
    private final KafkaProducer<Integer, String> producer;
    private final String topic;
    private final Properties props = new Properties();
    public UserKafkaProducer(String topic)
    {
        props.put("metadata.broker.list", "master2:6667");
        props.put("bootstrap.servers", "master2:6667");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
```



```
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        producer = new KafkaProducer<Integer, String>(props);
        this.topic = topic;
    }

    @Override
    public void run() {
        int messageNo = 1;
        while (true)
        {
            String messageStr = new String("Message_" + messageNo);
            System.out.println("Send:" + messageStr);
            producer.send(new ProducerRecord<Integer, String>(topic, messageStr));
            messageNo++;
            try {
                sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

消费 Kafka 消息:

```
public class UserKafkaConsumer extends Thread
{
    private final ConsumerConnector consumer;
    private final String topic;
    public UserKafkaConsumer(String topic)
    {
        consumer = kafka.consumer.Consumer.createJavaConsumerConnector(
            createConsumerConfig());
        this.topic = topic;
    }
    private static ConsumerConfig createConsumerConfig()
    {
        Properties props = new Properties();
        props.put("zookeeper.connect", "master1:2181,master2:2181");
```




```
        props.put("group.id", "group1");
        props.put("zookeeper.session.timeout.ms", "40000");
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");
        return new ConsumerConfig(props);
    }

    @Override
    public void run() {
        Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
        topicCountMap.put(topic, new Integer(1));
        Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.
createMessageStreams(topicCountMap);
        KafkaStream<byte[], byte[]> stream = consumerMap.get(topic).get(0);
        ConsumerIterator<byte[], byte[]> it = stream.iterator();
        while (it.hasNext()) {
            System.out.println("receive: " + new String(it.next().message()));
            try {
                sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

4.3 案例分析

微博广告业务涉及大量的日志数据回传、实时结算及实时流计算等业务。消息中间件是必不可少的，百亿级的数据传输对消息中间件的稳定性、吞吐量及横向扩展能力有很高的要求，消息中间件对于系统的横向扩展也起到重要的作用。

微博广告原有的日志传输采用 Scribe。Scribe 是 Facebook 开源的日志收集系统，在 Facebook 内部已经得到大量应用。它能够从各种日志源收集日志，存储到一个中央存储系统（可以是 NFS、分布式文件系统等）上，以便于进行集中统计、分析和处理。

Scribe Agent 与 Collector 之间的容错需要用户自己处理，并且在负载均衡方面不够好，资料相对较少，开源社区活跃度不高。相对于 Scribe，Kafka 支持的 Agent 相对比较丰富，开发使用比较简单。



4.3.1 日志采集

投放引擎进行日志的记录，采集工具进行日志的实时采集，然后实时地将日志写入 Kafka。微博广告的日志采集涉及广告请求日志、广告曝光日志、广告互动日志、广告负反馈日志等。对日志垂直分类，分为投放引擎、UVE、平台、MAPI、终端及第三方监控等环节。目前微博广告存在多个 Kafka 消息队列集群，针对不同的业务日志划分为不同的集群。日志收集使用了上百个 Kafka Topic。

4.3.2 实时结算

微博广告老结算系统采用 Scribe 日志结算模式，整体架构设计系统的处理能力偏弱，需要使用大量的机器。经过调研分析，将 Scribe 日志结算模式替换成 Kafka 进行系统的重新设计，在消费能力不足的情况下可以通过增加 Kafka 分区，提高系统的横向扩展能力。当遇到流量高峰或者突发性事件时流量会激增，消息队列的作用是：当系统处理消费能力不足时，可以通过增加分区、增加部署节点来提高系统吞吐量。当由于流量增加造成系统故障时，上游的数据可以写入 Kafka 集群，这样即使下游的服务宕机也不会造成结算数据的丢失，避免了对公司的收入造成一定的损失。Kafka 让系统与系统之间大大降低了耦合性。Kafka 支持同步和异步的消息写入，并且可以配置消息写入模式来降低丢失消息的风险，因为对于结算系统来说，消息丢失率高就会造成收入的减少。

4.3.3 实时计算

在互联网公司广告是核心业务，直接关系着公司的收入，因此广告投放系统的健康度及投放效果就显得非常重要。微博广告分为保量广告、竞价广告、效果广告等包含了超粉、品牌广告、涨粉、非粉、粉丝头条等产品。微博广告的请求量在百亿级，并且涉及的产品线较多，我们需要对微博产品各个产品线的请求、曝光、互动、结算等进行实时监控，也就是对海量数据进行实时计算。大量的数据需要实时地通过消息队列传输，实时计算引擎通过实时消费 Kafka 的数据进行计算。投放效果评估平台、微博广告实验平台同样需要实时地消费数据，进行投放算法的分析和反馈。

4.4 本章小结

本章介绍了 ZeroMQ、ActiveMQ、Kafka 等开源的消息队列的特点，重点对分布式消息队



列 Kafka 的组件概念、基本特性、安装与使用，以及 Java API 的简单使用进行了介绍。通过分布式消息队列在微博广告中的具体应用案例分析，让读者对分布式消息队列可以降低应用耦合度、提高应用可扩展性等特点有了更好的认识。

4.5 参考文献

[1] <https://kafka.apache.org/>



第 5 章

大数据存储技术

随着移动互联网的普及，用户越来越多，各个企业产生了大量的日志，这些日志主要包括 Apache、Nginx 等 WebService 产生的 access、error 日志，同时，各个业务系统还会产生大量的应用日志。这些日志一方面记录了系统的运行状态，另一方面还保留了系统业务处理的逻辑，不仅可以对系统进行实时监控，同时也为后续的问题跟踪和定位提供了数据基础，因此，运维很重要的一个工作就是将这些数据进行有效合理的存储。当数据规模比较小时，单机或者通过传统的关系型数据库就可以存储，然而数据一旦达到一定的规模（比如 TB、PB 级别），传统的数据存储方式就会存在很多问题，因此，针对大数据的存储诞生了 Hadoop 生态。本章将首先介绍传统存储方式及其利弊，然后详细介绍基于 HDFS 的分布式存储技术。

5.1 传统数据存储

5.1.1 传统应用的架构

我们首先来回顾一下 2010 年以前的互联网公司的大型论坛、SNS、博客等互动产品的技术架构，如图 5-1 所示为传统互动类应用架构示意图。其主要逻辑是：LVS 负责负载均衡；Squid/Varnish 承担图片、静态页的缓存功能；Nginx 用来做反向代理；Web 服务器处理业务逻辑；数据库存储业务数据；Cache 服务器主要用作对象缓存和列表缓存。由图 5-1 可见，其中存在大量能产生 access 日志的服务器，有效存储这些日志是一个难题。



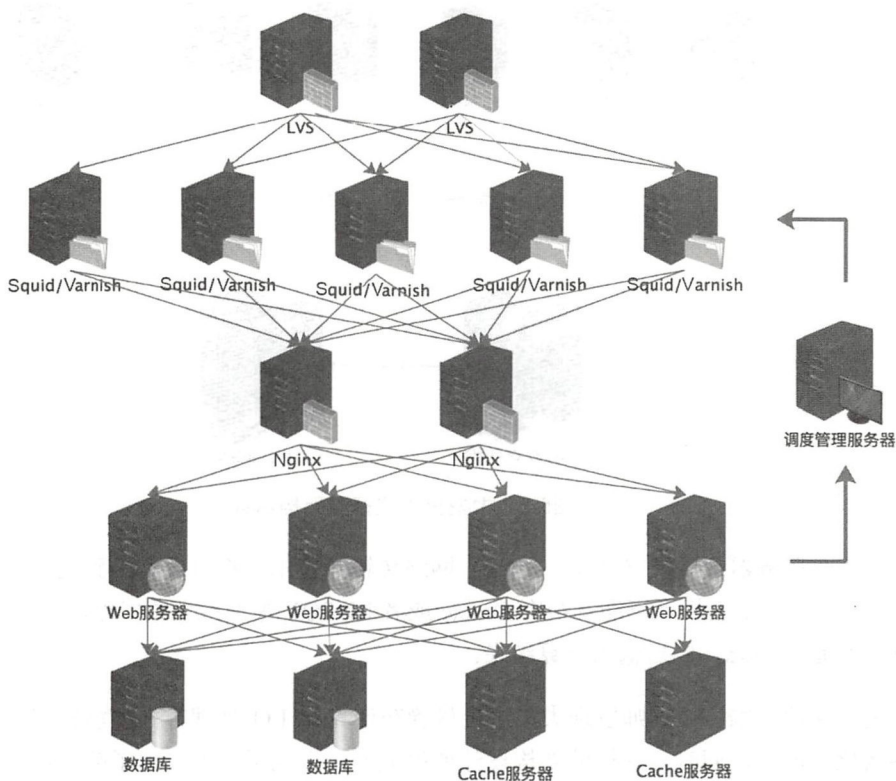


图5-1 传统互动类应用架构示意图

5.1.2 传统存储的运行机制

面对上面的业务系统产生的大量日志，传统的处理方式是采用集中存储。所谓集中存储就是指由一台大型主机或多台主机组成中心节点，数据集中存储于这个中心节点上，并且整个系统的所有业务单元都集中部署在这个中心节点上，系统所有的功能均由其集中处理。也就是说，在集中式系统中，每个终端或客户端仅仅负责数据的录入和输出，而数据的存储与控制处理完全交由主机来完成。

集中式存储最大的特点就是部署结构简单。由于系统往往基于底层性能卓越的大型主机，因此无须考虑如何对服务部署多个节点，也就不用考虑多个节点之间的分布式协作问题。共享一个文件系统及其他的物理设备资源，分配存储资源这种工作需要人工干预，由存储管理员来完成。这种集中式系统也是由关系型业务系统演变而来的，其架构示意图如图5-2所示。



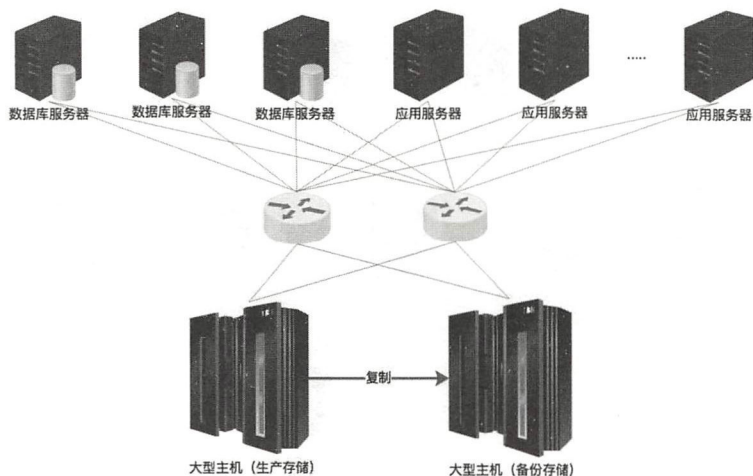


图5-2 大型集中式存储架构示意图

日志接收服务器通过网络共享、rsync、网络传输等技术，将日志集中到几台磁盘容量很大的大型主机上，大型主机上可能安装有 Oracle 或者 MySQL 等关系型数据库，用 Python、Perl、shell 来处理这些日志，然后输出计算结果。

通过日常的运维，工程师发现大型主机尽管在计算和 I/O 处理方面性能较强，但也避免不了设备老化的问题，同时，这样的架构设计使得大型主机成了单点。为了解决单点问题，通常的做法是使用一台具有相同配置的机器作为冷备，将两台机器耦合在一起，如果其中一台机器发生了故障，则可以快速地使用另一台机器顶替，维持业务的连续性，这就是最简单的集群形式。两台机器中总有一台一直处于备用状态。

5.1.3 传统存储带来的问题

问题 1：性能问题

由于数据量的激增，数据的索引效率也越来越为人们所关注。而动辄上 TB 的数据，甚至几百 TB 的数据，在索引时往往需要花上几分钟的时间。传统存储技术是把所有数据都当作对企业同等重要的数据来进行处理的，并且把所有的数据都集中存储到单一的存储体系之中，以满足业务的持续性需求。一旦数据达到一定规模，集中化存储就会成为数据读/写性能的瓶颈，在数据同时被读/写的过程中也会加重网络的负担，造成网络拥堵等性能问题。

问题 2：成本激增

大型主机也是非常昂贵的，通常一台配置较好的 IBM 大型主机，其售价达到上百万美元甚



至更高，因此也只有政府和金融、电信等企业才有能力采购大型主机。如果采用集中式存储方案，必然导致建设成本、管理成本、维护成本、能耗成本的急剧增加。计算技术不成熟，海量日志的价值没有被最大化挖掘出来，对存储的投资回报率不高，服务器越多，成本越高。

问题 3：单点问题

传统存储由于没有采用分布式文件系统，文件只在一台服务器上存储，当访问量大时，无法将压力平均分配到多个存储节点，因此在存储系统与计算系统之间存在着明显的传输瓶颈。比如单块磁盘 I/O 瓶颈，当单块磁盘损坏时，将面临数据丢失的情况，因此传统存储会带来一系列的单点问题。

问题 4：数据准确性

由于系统往往采用回滚写入方式，这种无序的频繁读/写操作，导致产生大量的磁盘碎片。随着使用时间的增加，将严重影响整个存储系统的读/写性能，甚至导致存储系统被锁定为只读状态，而无法写入新的数据。当有大量写入时，共享磁盘、rsync 等技术在文件切割、断点续传等方面显得力不从心，经常会出现日志文件不完整的情况，影响了整体数据的准确性。与传统存储相比，基于分布式的云存储则具有量身定制、成本低、管理方便，还能满足存储需求等突出优势。

2009 年，阿里巴巴集团发起了一项“去 IOE”运动。背景是阿里巴巴集团从 2008 年开始各项业务都进入了井喷式的发展阶段，这对于后台 IT 系统的存储能力提出了非常高的要求，一味地针对小型机和高端存储进行不断扩容，无疑会产生巨大的成本。同时，集中式存储存在诸多问题，无法完全满足互联网应用爆炸式的发展需求。因此，为了应对业务快速发展给 IT 系统带来的巨大挑战，从 2009 年开始，阿里巴巴集团启动了“去 IOE”计划，其电商系统正式迈入了分布式系统时代。后续章节将重点介绍分布式存储技术。

5.2 基于HDFS的分布式存储

5.2.1 分布式存储的定义

分布式存储就是指通过网络连接每台机器，使这些分散的存储资源构成一个虚拟的存储设备，数据分散在每台机器上的各个角落。与集中式存储技术不同，分布式存储技术并不是将数据存储在某个或多个特定节点上的。

分布式存储的出现是因为在互联网时代信息数据大爆炸，传统的集中式存储已经难以满足

大型应用的数据存储需求。

按数据存储模型来划分，分布式存储服务分为：

文件模型——对应分布式文件系统，如 GFS、HDFS。

关系模型——对应分布式数据库系统，如 Google Spanner、OceanBase、ClickHouse。

键值模型——很多 NoSQL 系统采用键值模型，如 Redis、Memcache。

对于分布式存储，需要关注如下几点。

- 数据的分布和负载均衡。
- 存储系统的容错问题。
- 系统的可扩展性。
- 如何保障可靠性和准确性。
- 性能和容错能力。

分布式存储能很好地解决上述问题。当然，集中式存储也能解决上面的问题，在集中式存储中存在独立磁盘冗余阵列（Redundant Array of Independent Disk, RAID）技术，它把相同的数据存储到多块硬盘的不同位置。数据可以通过磁盘阵列控制程序均匀分布在多块硬盘上，这样就解决了负载均衡问题，通过冗余解决了可靠性问题。虽然磁盘阵列解决了单一磁盘的脆弱性问题，但是并不能提升整体的容错能力和可用性。同理，扩展能力受制于物理扩展槽的数量。

5.2.2 HDFS的基本原理

HDFS^[1]（Hadoop Distributed File System）是 Apache 基金会下的 Hadoop 项目的一个主要组成部分，它被设计成适合运行在通用硬件（Commodity Hardware）上的分布式文件系统。它和现有的分布式文件系统有很多共同点，同时它和其他的分布式文件系统的区别也是很明显的。HDFS 是一个具有高度容错性的系统，适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问，非常适合基于大规模数据集的应用。在最开始时 HDFS 是作为 Apache Nutch 搜索引擎项目的基础架构而开发的。HDFS 是 Apache Hadoop Core 项目的一部分。HDFS 具有高容错性（fault-tolerant）的特点，并且被设计用来部署在低廉的硬件上。它提供了高吞吐量（High Throughput）来访问应用程序的数据，适合那些有着超大数据集的应用程序。HDFS 放宽了 POSIX

约束，这样可以实现以流的形式访问文件系统中的数据。

HDFS 的主要组成部分：NameNode、SecondaryNameNode、DataNode。HDFS 的基本操作：读、写、均衡。

5.2.3 HDFS架构解析

HDFS 架构包含三个部分，即 NameNode、DataNode 和 Client，每个部分都有清晰的职责划分。如图 5-3 所示为 HDFS 架构图^[2]。

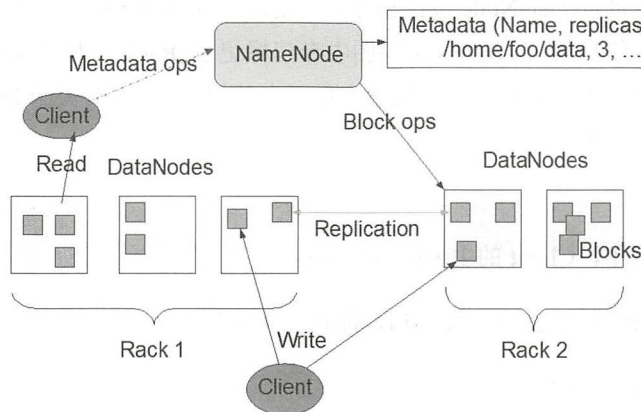


图5-3 HDFS架构图（图片来源：HDFS官方网站^[2]）

从图 5-3 中可见，HDFS 采用的是中心总控式架构，NameNode 就是集群的中心节点。

1. NameNode

NameNode 的主要职责是管理整个文件系统的元信息（Metadata）。元信息主要包括：

- HDFS 类似于单机文件系统，以目录树的形式组织文件，称为 File system namespace。
- 文件副本数，是针对每个文件设置的。
- 文件块到数据节点的映射关系。

在图 5-3 中，指向 NameNode 的 Metadata ops 主要就是针对文件的创建、删除、读取和设置文件的副本数等操作的，因此所有的文件操作都绕不过 NameNode。除此之外，NameNode 还负责管理 DataNode，例如新的 DataNode 加入集群、旧的 DataNode 退出集群、在 DataNode 之间负载均衡文件数据块的分布等。

2. DataNode

DataNode 的主要职责如下：

- 存储文件块。
- 响应 Client 的文件读/写请求。
- 执行文件块的创建、删除和复制操作。

从图 5-3 中可以看到，有一个 Block ops 的操作箭头从 NameNode 指向 DataNode，但实际上 NameNode 不会主动向 DataNode 发出指令，而是通过 DataNode 定期向 NameNode 发送心跳来携带回传的指令信息的。在 HDFS 架构图中专门标记了 Rack 1 和 Rack 2，这表明 HDFS 在考虑文件数据块的多副本分布时针对机架感知做了专门设计。更多的关于 DataNode 的设计实现分析，大家可以参阅相关文档。

3. Client

在 HDFS 交互过程中，Client 的主要职责如下：

- 提供面向应用程序语言的一致 API，简化应用编程。
- 改善 HDFS 的访问性能。

Client 能够改善性能，原因是针对读可以提供缓存（Cache），针对写可以通过缓冲（Buffer）批量方式。

5.2.4 HDFS的优势

我们知道 Hadoop 是为了处理大数据而诞生的一个系统，而 HDFS 是为了存储大数据而产生的一个分布式文件系统，所以它在设计上就考虑了很多大数据处理存储的一些特点。下面我们介绍 HDFS 在设计上的优势。

第一，解决硬件错误问题。硬件错误是常态而不是异常。HDFS 可能由成百上千台服务器构成，每台服务器上都存储着文件系统的部分数据。大规模服务器构建的复杂系统，其任何一个组件都有可能失效，这意味着总是有一部分 HDFS 组件是不工作的。因此，错误检测和快速、自动恢复是 HDFS 最核心的架构目标。

第二，流式数据访问机制。运行在 HDFS 上的应用和普通的应用不同，需要流式访问它们

的数据集。HDFS 的设计更多地考虑到了数据批处理，而不是用户交互处理。相比于数据访问的低延迟问题，更关键的是数据访问的高吞吐量。POSIX 标准设置的很多硬性约束对 HDFS 应用系统不是必需的，为了提高数据的吞吐量，在关键方面对 POSIX 的语义做了一些修改。

第三，软硬件平台间的移植。HDFS 非常容易在平台间迁移，这也是 HDFS 被很多大型应用广泛采用的主要原因。

第四，支持大规模数据集。HDFS 被设计成支持大文件存储，一般运行在 HDFS 上的应用都具有很大的数据集。HDFS 上的一个典型文件大小一般都在 GB 至 TB 级别。HDFS 能在一个集群中扩展到数百个节点，一个单一的 HDFS 实例能支撑数以千万计的文件。

第五，简单的一致性模型。大数据的应用经常需要一个“一次写入，多次读取”的文件访问模型。一个文件经过创建、写入和关闭之后就不需要改变了。这一设计解决了数据一致性问题，并且使高吞吐量的数据访问成为可能。Map/Reduce 应用或者网络爬虫应用都非常适合采用这个模型。

第六，“移动计算比移动数据更划算”。做过 HTTP 的高性能 Web 产品的技术人员都知道，一个请求的计算离它操作的数据越近越高效，当数据达到海量级别的时候更是如此。原因是这样就能降低网络阻塞的影响，提高系统数据的吞吐量。将计算移动到应用附近，比将数据移动到应用所在显然更好。HDFS 为应用提供了将它们自己移动到数据附近的接口。

注：上面 6 个特点的描述主要引用自 Hadoop 1.0.4 官方文档，文档中有些介绍已经过时，上面已经进行了修正。

总结：HDFS 被设计用来存储海量大文件，它对硬件并不挑剔，普通的服务器就可以，而且支持异构；HDFS 所专注的场景主要是流式的，即写一次，但读很频繁的模式；HDFS 专注的是提高整体系统的吞吐量。

5.2.5 HDFS不适合的场景

HDFS 在大数据存储方面有着卓越的优势，但它并不是万能的，下面总结了 HDFS 不太适合的几个场景。

第一，低延迟访问的场景。HDFS 主要是针对提高系统吞吐量做了很多设计优化，而不是数据访问的实时性。如果是对访问的及时性有较高要求的场景，则可考虑使用 HBase、Druid 等开源软件。

第二，较多小文件的存储。HDFS 是用来存储海量大文件的，不太适合存储大量小文件。原因是 HDFS 的 NameNode 是将 HDFS 上的元数据存储在内存中的，从而使得限制 HDFS 存储量的就是机器内存的大小。根据经验值，一个文件、目录或者块（block）的元数据大概会占用 150 字节的内存，也就是说，如果现在有 100 万个文件，每个文件占一个块，那么就需要大概 300MB 的内存。目前来看，对大量小文件的存储还是用一些对象存储系统比较好。

第三，有多个写入者，以及随机修改的场景。现在的 HDFS 不支持同时有多个写入者，而且仅支持追加写（append-only）模式，不支持从任意地方进行数据修改。这个场景后续可能会支持，但就算支持了，相对来说，也不会非常高效。

5.3 分层存储

上面讲了基于 HDFS 的分布式存储，很多企业特别是互联网企业在 HDFS 上都搭建了 Hive 数据库，用来存储结构化数据。随着企业的数据种类越来越多，数据量级越来越大，设计什么样的存储，以及如何高效地管理这些数据就成了重要的问题。因此，数据仓库和仓库分层存储等技术得以飞速发展。本节将重点介绍大数据环境下的数据仓库和分层存储技术。

5.3.1 数据仓库

数据仓库之父比尔·恩门（Bill Inmon）在 1991 年出版的 *Building the Data Warehouse*（《建立数据仓库》）一书中提出了数据仓库的定义并被广泛接受——数据仓库（Data Warehouse）是一个面向主题的（Subject Oriented）、集成的（Integrated）、相对稳定的（Non-Volatile）、反映历史变化（Time Variant）的数据集合，用于支持管理决策（Decision Making Support）。

数据仓库是一个过程而不是一个项目，数据仓库是一个环境而不是一个产品。数据仓库为用户提供用于决策支持的当前和历史数据，这些数据在传统的操作型数据库中很难或不能得到。数据仓库技术是为了有效地把操作型数据集成到统一的环境中，以提供决策型数据访问的各种技术和模块的总称。其所做的一切都是为了让用户更快、更方便地查询所需要的信息，提供决策支持。

1. 数据仓库的特点

（1）面向主题

操作型数据库中的数据组织是面向事务来处理任务的，各个业务系统之间相互分离，而数

据仓库中的数据是按照一定的主题域进行组织的。

(2) 集成

数据仓库中的数据是在对原有分散的数据库数据抽取、清理的基础上经过系统加工、汇总和整理得到的,必须消除源数据中的不一致性,以保证数据仓库内的信息是关于整个企业的一致性的全局信息。

(3) 相对稳定

数据仓库中的数据主要供企业决策分析之用,所涉及的数据操作主要是数据查询,一旦某个数据进入数据仓库以后,一般情况下将被长期保留。也就是说,在数据仓库中一般存在大量的查询操作,但修改和删除操作很少,通常只需要定期加载、刷新即可。

(4) 反映历史变化

数据仓库中的数据通常包含历史信息,系统记录了企业从过去某一时间点(如开始应用数据仓库的时间点)到目前的各个阶段的信息,通过这些信息可以对企业的发展历程和未来趋势做出定量分析和预测。

2. 数据仓库的组成

(1) 数据仓库数据库

数据仓库的数据库是整个数据仓库环境的核心,是存放数据的地方,提供对数据检索的支持。相对于操作型数据库来说,其突出的特点是对海量数据的支持和快速检索。

(2) 数据抽取工具

数据抽取工具把数据从各种各样的存储方式中拿出来,进行必要的转换、整理,再存放到数据仓库内。对各种不同的数据存储方式的访问能力是数据抽取工具的关键。数据转换包括:删除对决策应用没有意义的数据段、转换为统一的数据名称和定义、计算统计和衍生数据、给缺值数据赋缺省值、统一不同的数据定义方式。

(3) 元数据

元数据是描述数据仓库内数据的结构和建立方法的数据。按用途可将元数据分为两类:技术元数据和商业元数据。

技术元数据是数据仓库的设计和管理人员用于开发和日常管理数据仓库使用的数据。技术

元数据包括：数据源信息、数据转换的描述、数据仓库内对象和数据结构的定义、数据清理和数据更新使用的规则、源数据到目的数据的映射、用户访问权限、数据备份历史记录、数据导入历史记录、信息发布历史记录等。

商业元数据从商业业务的角度描述了数据仓库中的数据。商业元数据包括：业务主题的描述，以及所包含的数据、查询、报表。

元数据为访问数据仓库提供了一个信息目录，这个目录全面描述了数据仓库中都有什么数据、这些数据是怎么得到的，以及怎么访问这些数据。它是数据仓库运行和维护的中心，数据仓库服务器利用它来存储和更新数据，用户通过它来了解和访问数据。

（4）访问工具

访问工具为用户访问数据仓库提供手段。访问工具有数据查询和报表工具、应用开发工具、联机分析处理（OLAP）工具、数据挖掘工具。

（5）数据集市（Data Market）

数据集市是为了特定的应用目的或应用范围，而从数据仓库中独立出来的一部分数据，也可称为部门数据或主题数据。在数据仓库的实施过程中往往可以从一个部门的数据集市着手，以后再用几个数据集市组成一个完整的数据仓库。需要注意的是，在实施不同的数据集市时，具有相同含义的字段定义一定要相容，这样在以后实施数据仓库时才不会造成大麻烦。

上述内容是大家对数据仓库的普遍认知，它在很早之前就存在，并不是在大数据出现之后才有的。在大数据时代，我们对数据仓库的理解是，它代表的是一种对数据的管理和使用的方式，是一套包括 ETL、调度、建模在内的完整的理论体系。

5.3.2 数据仓库分层架构

数据仓库采用分层架构，分为缓冲层、操作数据层、明细数据层、汇总数据层和数据集市层。如图 5-4 所示为数据仓库分层架构示意图。

1. 缓冲层（Buffer）

- 概念：用于存储每天的增量数据和变更数据。
- 数据生成方式：直接从 Kafka 接收源数据或从业务库抽取，需要业务表每天生成 update、delete、insert 数据，只生成 insert 数据的业务表，数据直接接入操作数据层。

析的公共资源。

- 数据生成方式：在操作数据层清洗或 JOIN 维度表之后生成。
- 日志删除方式：长久存储。
- 表 Schema：一般按天创建分区，没有时间概念的按具体业务选择分区字段。
- 库与表命名：库名为 dwd；表名，初步考虑格式为 dwd_业务表名。
- 旧数据更新方式：直接覆盖。

4. 汇总数据层（DWS, Data Warehouse Summary）

- 概念：在数据仓库中明细数据层和数据集市层之间的一个过渡层次，对明细数据层的生产数据进行轻度综合和汇总统计。汇总数据层与明细数据层的主要区别在于二者的应用领域不同，明细数据层的数据来源于生产型系统，并为满足一些不可预见的需求而进行沉淀；汇总数据层则面向分析型应用进行细粒度的统计和沉淀。
- 数据生成方式：由明细数据层按照一定的业务需求生成轻度汇总表。明细数据层需要复杂清洗的数据和需要 MR 处理的数据也经过处理后接入汇总数据层。
- 日志删除方式：长久存储。
- 表 Schema：一般按天创建分区，没有时间概念的按具体业务选择分区字段。
- 库与表命名：库名为 dws；表名，初步考虑格式为 dws_日期_业务表名。
- 旧数据更新方式：直接覆盖。

5. 数据集市层（DM, Data Market）

- 概念：数据集市层又称为数据集市或宽表。按照业务划分，如流量、订单、用户等，生成字段比较多的宽表，用于提供后续的业务查询、OLAP 分析、数据分发等。
- 数据生成方式：由汇总数据层和明细数据层的数据计算生成。
- 日志删除方式：长久存储。
- 表 Schema：一般按天创建分区，没有时间概念的按具体业务选择分区字段。
- 库与表命名：库名为 dm；表名，初步考虑格式为 dm_业务表名。

- 旧数据更新方式：直接覆盖。

5.3.3 分层存储的好处

总结起来，数据仓库分层有如下好处。

第一，数据结构更明确。分层之后，每一层都有其作用域，这样我们在使用表的时候，能更方便地理解，提高工作效率。

第二，数据血缘追踪，便于管理。数据团队产出的是一个能直接使用的业务表，这些表的来源有很多，假如某一个来源表出问题了，但有了分层，数据血缘关系更清晰，它能够帮助我们快速准确地定位到问题，并清楚问题的影响范围。

第三，复杂问题简单化。分层之后，一个复杂的任务被分解成多个步骤来完成，每一层只处理单一的步骤，比较简单和容易理解。而且便于维护数据的准确性，当数据出现问题之后，可以不用修复所有的数据，只需要从有问题的步骤开始修复即可。

第四，表共用，减少了重复计算。由于数据仓库的分层设计，通过开发一些共用的中间层数据表，进而减少了重复计算，节省了计算资源。

第五，屏蔽原始数据的异常和业务变更的影响。比如，当业务发生变更（如对日志增加了一个新的字段）时只会影响最底层的数据仓库表结构，不会影响上层的数据业务层（如数据集市层），应用方对底层的数据异常无感知，从而降低了应用方因数据异常和业务变更带来的风险。

5.4 案例分析

微博广告每天会产生 10TB 以上的增量数据，因为微博广告业务复杂，涉及的数据种类较多，主要分为广告订单、广告受众、广告主、广告内容等主题。本节将结合微博广告数据的应用，讲解微博广告数据存储架构，重点以 Kimball 的建模步骤为例讨论常用的数据仓库建模方法，最后介绍存储压缩及数据安全性问题。

5.4.1 数据存储架构

微博广告数据架构采用业内通用的 Lambda 架构，实时计算和离线批处理计算同时运行。整个架构分为采集层、计算层、服务层和数据应用层，数据根据需求在各层之间流动。这里对数据架构图不进行详细介绍，重点是根据数据架构推导出存储架构。如图 5-5 所示为数据存储

架构图，分为业务存储、消息存储、离线存储、分析存储和应用存储。

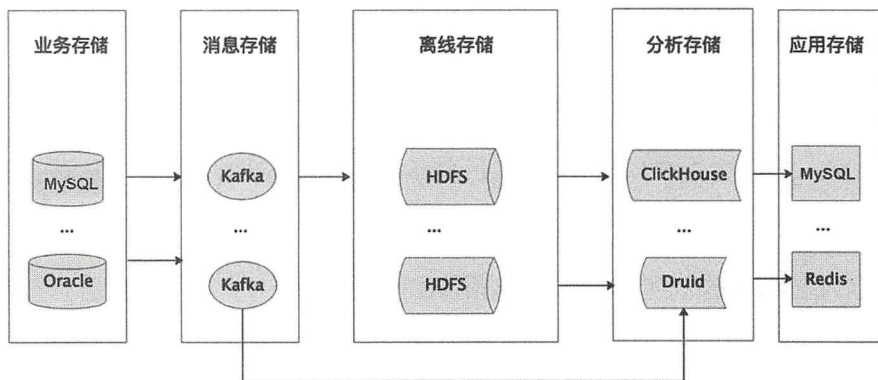


图5-5 数据存储架构图

业务存储：指业务的数据库，通常是业务系统的备份库或从库。使用 MySQL 或 Oracle 存储，如广告业务系统里的订单库。

消息存储：有两个来源，一是业务的数据库消息数据，一条记录一个消息；二是结合数据架构，日志采集工具也会产生消息数据。这两种消息数据都会存在 Kafka 上，在一定的时间之后自动过期。

离线存储：基本上所有的数据都要在 HDFS 上做离线存储。

分析存储：微博广告数据主要用 ClickHouse 来支撑离线的分析数据存储，应用于多维度的数据钻取。Druid 支持实时数据存储，用于监控和满足实时需求。

应用存储：主要存储结果集数据，服务于 BI 报表和数据应用产品。可以使用 MySQL 或 Redis 支持。

另外，存储模型的设计借用的是业界通用的 Kimball 和 Inmon 模型。

5.4.2 数据仓库建模

Kimball 和 Inmon 是两种主流的数据仓库方法论，分别由 Ralph Kimball 和 Bill Inmon 提出。在实际的数据仓库建设中，业界往往会相互借鉴使用两种开发模式。本节将详细介绍 Kimball 和 Inmon 理论在实际数据仓库建设中的应用与对比，通过数据仓库理论武装数据仓库实践。本文参考了《数据仓库（第 4 版）》^[3]中的一些定义。

1. 什么是维度模型（Kimball）

（1）基本概念

Kimball 模型的设计流程是自底向上的，即从数据集市到数据仓库再到数据源（先有数据集市，后有数据仓库）的一种敏捷开发方法，跟数据流向正好相反，图 5-6 展示了数据流向。对于 Kimball 模型，数据源往往是给定的若干个数据库表，需要从这些 OLAP（On-Line Analytical Processing，联机分析处理，是数据仓库的主要应用，主要用于支持复杂的分析决策）产生的事务型数据表中抽取分析型数据结构，再放入数据集中以方便下一步的 BI 与决策支持。

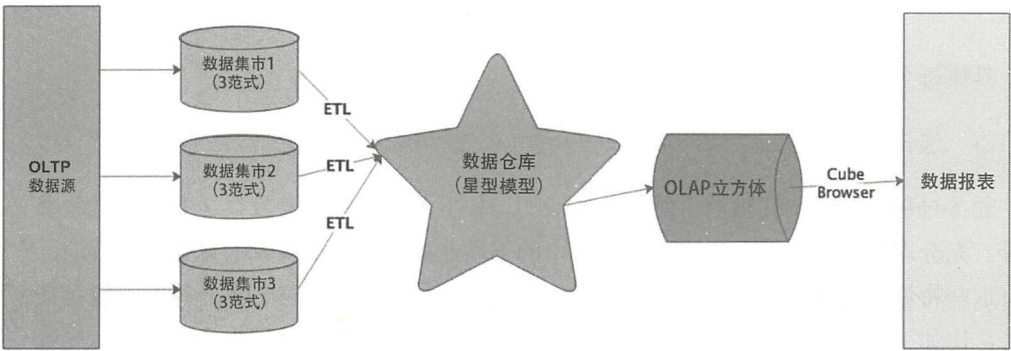


图5-6 Kimball模型中的数据流向图

（2）模型流程

通常，Kimball 是以最终任务为导向的。首先，从数据报表层得到数据需求，并确认目标。然后，尝试将数据按照目标先拆分成不同的表需求，再根据表需求，确认 OLAP 上的数据立方体。接下来，从 OLAP 数据源中抽取满足 3 范式的表。最后，从 OLAP 立方体的角度出发，结合抽取出来的满足 3 范式的表，使用星型模型结构，设计出数据仓库，从而完成整个过程。

Kimball 往往意味着快速交付、敏捷迭代，不会对数据仓库架构做过多复杂的设计，在变幻莫测的互联网行业，这种架构方式逐渐成为一种主流范式。

（3）Kimball 建模

Kimball 建模方法的精髓是简单、实用，建模的四个步骤是核心，如图 5-7 所示。



图5-7 Kimball建模步骤

具体每个步骤的执行过程如下。

○ 选择业务过程

业务过程是源系统提供的自然的业务活动，如下单、支付、发放代金券、点击、曝光、请求等。充分理解它们，有助于辨别源系统中的不同业务过程。事实表的设计依赖活动，它一般具有这些特性：用行为动词表示业务过程的活动，比如下单、支付、曝光等；一般由某个平台支持，比如下单由交易平台支持、曝光由交互平台支持等；业务过程中含有度量，度量一般由操作过程直接生成，比如用户支付金额、曝光次数等。

我们既需要理解什么是业务过程，也需要理解什么不是业务过程，这样才能取舍。比如不同部门的功能划分就不是业务过程，我们应该将注意力放在业务过程上，而不是不同的部门上，这样才能避免重复地获取数据。

○ 声明粒度

粒度是描述表中一行记录代表的含义。它包含的是与表中度量值相关的细节所达到某种程度的信息。比如订单事实表，粒度为某一天的某个订单。这一步特别容易被忽略，粒度声明需要达成共识，否则极有可能到选定维度和确定事实时要返工重来。另外，在开发过程中特别容易用维度列代替粒度声明。

○ 选定维度

描述业务过程中度量的事件数据，维度用来描述：谁、什么事、何时何地、为何、如何。比如订单事实表的维度是：产品、供应商、下单人、订单状态、退款状态等。

○ 确定事实

确定事实是确定业务过程的度量指标，如指标何来、哪些指标必须保留、哪些指标必须删除、待定指标如何处理等，必须综合考虑业务用户需求和现实数据的实际情况。粒度和事实是有区别的，粒度说明了每一行代表什么意思，而事实是里面包含哪些列，比如下单金额、退款金额、购买份数等。

○ 选择冗余维度（建议使用）

本步骤在图 5-7 中没有列出来，但在实际应用的过程中，我们发现事实表是需要做适度冗余的。Kimball 之所以要减少冗余，是因为当时的存储成本较高，需要减少存储消耗。特别是在互联网企业中，需要考虑下游用户的使用效率，降低获取数据的复杂性，减少关联表的数量，所以需要做冗余，比如供应商名称、产品名称、品类名称等。

2. 什么是E-R模型（Inmon）

（1）基本概念

Inmon 模型从流程上看是自顶向下的，即从数据源到数据仓库再到数据集市（先有数据仓库，后有数据集市）的一种瀑布流开发方法，如图 5-8 所示。对于 Inmon 模型，数据源往往是异构的。因此，这里的数据处理工作主要集中在对异构数据的清洗上，包括数据类型检验、数据值范围检验，以及其他一些复杂规则。在 Inmon 模型中，并不强调事实表和维度表的概念，因为数据源变化的可能性较大，而需要更加强调数据的清洗工作，从中抽取实体和关系。

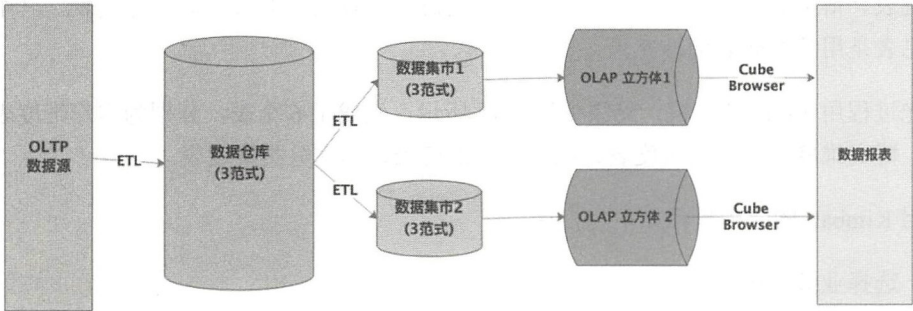


图5-8 Inmon模型流程图

（2）模型流程

通常，Inmon 是以数据源为导向的。首先，需要探索性地从 OLAP 数据源中获取尽量符合预期的数据，尝试将数据按照预期划分为不同的表需求。然后，将这些表划分成实体-关系模型，

再在实体-关系模型的基础上细化主题表的数据项，满足 3 范式，形成数据仓库。接下来，根据具体的需求，并考虑性能和平台的特点，进行合并或分区的设计。最后，形成 OLAP 上的数据立方体，并输出到 BI 系统中辅助具体业务。

维度模型（Kimball）和 E-R 模型（Inmon）比较如图 5-9 所示。

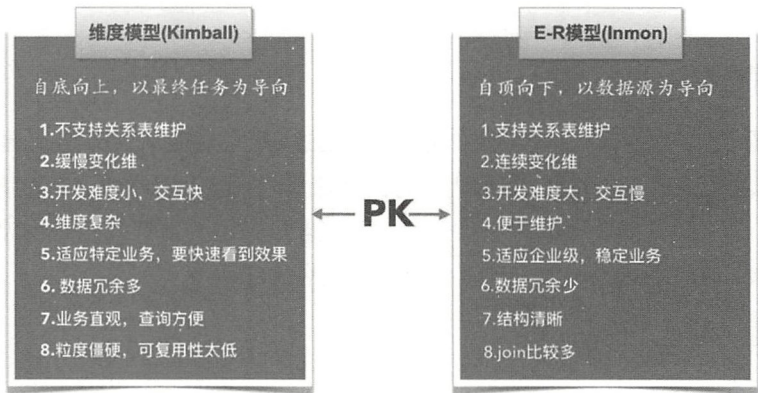


图5-9 维度模型（Kimball）和E-R模型（Inmon）比较

3. 建模实例

上面介绍了 Kimball 和 Inmon 两种数据建模的方法，接下来我们以微博广告的实际应用，微博广告博文曝光的过程，描述具体建模的流程。通过分析可知，曝光的过程有两个实体，即用户和博文，这两个实体产生曝光的关系。因此，用 Inmon 模型生成如图 5-10 所示的表，包括：用户基础表、博文表、曝光关系表、用户详细信息表、频道字典表、博文分类表。其中，用户详细信息表是用户基础表的补充。

曝光过程用 Kimball 建模，结果如图 5-11 所示，生成了 6 个表，分别为用户维度表、曝光事实表、博文维度表、频道维度表、微博类型维度表、博文分类维度表。

套用 Kimball 的建模步骤如下。

（1）选择业务过程

微博博文曝光，这个业务过程比较简单和明确。

（2）声明粒度

某个用户对某个博文在某一时刻产生的一次曝光。如果一个用户对一个博文看了多次，就有多条记录。

E-R模型 (inmon)

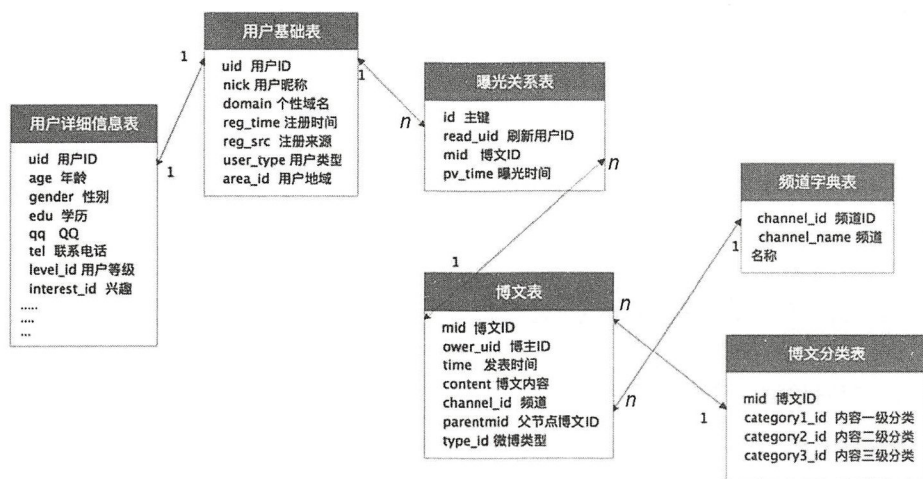


图5-10 Inmon模型实例

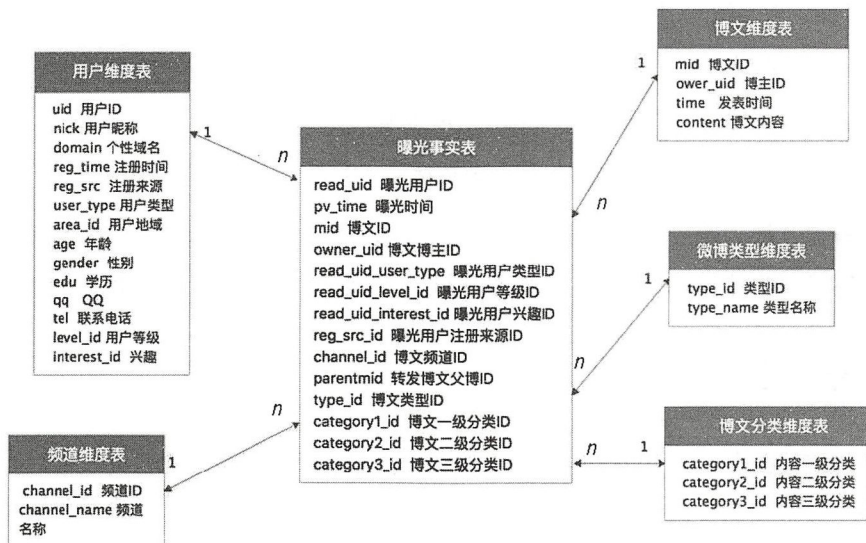


图5-11 Kimball模型实例

(3) 选定维度

维度有：用户维度、博文维度、微博类型维度等常规维度，可以满足常规的查询需求。

(4) 确定事实

事实的度量指标为曝光次数、曝光用户数。

(5) 选择冗余维度

冗余维度有博文分类维度、博文频道 ID 等。冗余多少，是根据具体的需求来定的。例如：如果需求是从曝光事实表中查询曝光用户的年龄或性别分布，那么可以把用户维度表中的年龄和性别字段代入曝光事实表中，虽然产生了冗余，但查询非常方便。

5.4.3 常见的存储问题及解决方案

问题 1：存储数据的安全问题

处理：关于数据安全问题，是很多公司早期容易忽视的。微博广告在存储安全这方面找到了解决办法。第一，实行严格的库表权限管理；第二，数据安全分等级，对于特别重要的数据，在入库前就对数据进行加密，在数据加工过程中一直是密文，只有在需要使用时才及时按需解密。

问题 2：文件太多，存储不了

处理：合并小文件，然后压缩，压缩，还是压缩。

问题 3：存储结构越来越混乱，三年后无法维护

处理：数据管理工作从第一个表的设计开始就要着手，重视元数据和主数据的管理，数据命名、输入输出规范越早建立越好。不要忽视每一个细节，不要轻视每一个很小的工作。

5.5 本章小结

本章从传统数据存储到 HDFS 分布式存储详细介绍了大数据存储的相关技术，并介绍了数据仓库及其分层设计思路。最后以微博广告的实际应用作为案例，介绍了数据存储架构和建模方法。

5.6 参考文献

[1] HDFS Architecture. <http://hadoop.apache.org/>

[2] Robert Chansler, Hairong Kuang, Sanjay Radia, Konstantin Shvachko, and Suresh Srinivas. The Hadoop Distributed File System

[3] William H. Inmon 著. 王志海等译. 数据仓库 (原书第 4 版). 北京: 机械工业出版社, 2006 年 8 月

第6章

大规模数据离线计算分析

在很多互联网公司，离线计算其实就是大数据计算的代名词。离线计算覆盖了 90% 的数据需求，公司的业务越复杂，离线计算的场景就越复杂。离线计算的特点是在计算开始前已知所有的输入数据，这些数据不会在计算的过程中发生变化，且在计算完后就要立即得出结果。

本章我们先回顾一下 Linux 神级工具 sed 和 awk，以及 Pandas，然后详细讲述 MapReduce 的执行过程，以及数据倾斜的问题，最后阐述离线计算在微博广告中的应用。

6.1 经典的离线计算

本节讨论的经典的离线计算是指非分布式环境下的离线数据处理技术。经典的离线计算指单机计算，大部分是单机下的文本处理和查询。

6.1.1 Linux神级工具sed和awk

Linux 系统的两个神级工具 sed^[1]和 awk^[1]是 Linux 高手们必备的利器，很值得我们研究。

1. 什么是sed

关于 sed，《sed 与 awk》中（见 1.2 节）的解释是：sed 本质上是一个编辑器，但它是非交互式的，这一点与 vim 不同；同时它又是面向字符流的，输入的字符流经过 sed 的处理后输出。这两个特性使得 sed 成为命令行下非常实用的一个处理工具。

sed 本身也是一个管道命令，主要用来进行关键字的分析和统计。此外，它还可以对数据进行替换、删除、新增，以及特定行的选取等操作。

2. sed使用案例

案例 1:

```
nl /etc/passwd | sed '2,5d'
```

说明:

- sed 参数'2,5d'中的 d 表示删除，数字用于范围选择，命令运行的结果就是删除了第 2~5 行。
- 命令中带上参数-i，表示修改源文件内容。
- 参数-e 表示直接在命令行中执行编辑操作，该参数是缺省值。
- sed 后面接的动作，请务必使用单引号（'）围起来。
- 如果只想删除第 2 行，那么命令就是：nl /etc/passwd | sed '2d'。
- 如果想删除第 3 行到最后一行，则可以这样写：nl /etc/passwd | sed '3,\$d'。

注：\$表示最后一行。

案例 2:

```
nl /etc/passwd | sed '2a drink tea'
```

说明:

- 命令运行的结果就是在第 2 行的后面加上了“drink tea”字样。
- 如果想在第 2 行的前面加上字符串，则可以这样写：nl /etc/passwd | sed '2i drink tea'。

注：2a 中的 a 是指第 2 行的后面，而 2i 中的 i 则是指第 2 行的前面。

3. 什么是awk

简单来说，awk 就是一个数据处理工具。相比于 sed 常常用于一整行的处理，awk 则比较倾向于将一行分成多个“字段”来处理。因此，awk 适合处理小数据集。结合 awk 强大的语法，它可被广泛应用于各种计算和数据处理任务中。

4. awk的主要功能

- 将文本文件看作由记录和字段组成的文本数据库。
- 使用变量操作数据库。

- 使用算术和字符串操作符。
- 使用普通的程序设计结构，例如循环和条件结构。
- 生成格式化报表。
- 定义函数。
- 从脚本中执行 UNIX 命令。
- 处理 UNIX 命令的结果。
- 处理命令行的参数。
- 处理多个输入流。

5. 基本语法

`awk '条件类型 1{动作 1} 条件类型 2{动作 2} ...' filename`

- `awk` 后面接对数据进行的处理动作，要使用单引号和大括号来设置作用域。
- `awk` 可以处理紧跟着的文件(`filename` 参数)，也可以读取来自上一个命令的标准输出。
- `awk` 主要处理每一行的字段内数据，默认的字段分隔符为空格。

6. `awk`的处理流程

- (1) 读入第 1 行，并将第 1 行的数据填入 `$0`, `$1`, `$2` 等变量中。
- (2) 根据条件类型的限制，判断是否需要执行后面的动作。
- (3) 执行所有的动作与条件类型。
- (4) 若还有后续的“行”数据，则重复第 1~3 步，直到所有的数据都读完为止。

更多的关于 `sed` 和 `awk` 技术的内容，读者可以参阅《`sed` 与 `awk`》^[1]。

6.1.2 Python数据处理Pandas基础

1. Pandas简介

经过多年的发展，Pandas^[2]已经成为Python处理数据时最常使用的Package。最开始时Pandas是为处理数据而开发的，一般当处理的数据量小于5TB时，使用PythonPandas足以应对。

2. Pandas 的数据结构

(1) Series

Series 是一个一维的 array-like 对象。它由两部分组成：任意 numpy 数据类型的 array 和数据标签（称之为 index）。因此，一个 Series 有两个主要参数，即 values 和 index。通过传递一个能够被转换成类似于序列结构的字典对象来创建 Series，字典的 key 用 index 表示。

(2) DataFrame

DataFrame 可以用来表示图表类型、数据库关系类型的数据，它包含数个顺序排列的 column，同一个 column 中的数据类型是一致的，但是不同 col 中的数据类型可以不一致。DataFrame 有两个 index，即 row 和 column。通过同等长度的 list 或者 array、dictionary 等来创建 DataFrame。

通过 DataFrame 处理数据的常见函数如下。

- sort()函数：返回一个排序好的对象集合。默认升序排列，降序排列设置 ascedning 参数为 false。
- sort_index()函数：按照行或者列进行排序。默认按照行排序，当设置 axis=1 时即按照列排序。默认升序排列，降序排列设置 ascedning 参数为 false。
- sort_values()函数：按照 value 进行排序。
- is_unique()函数：用于判断是否唯一，返回 true 或 false。
- unique()函数：用于返回不重复的值，返回一个数组。

通过 DataFrame 选择数据可以使用 selection 方法、切片（slicing）和索引（index），具体操作方法如下：

- 选择一个单独的列，将会返回一个 Series，df['A']和 df.A 是一个意思。
- 通过“[]”选择，将会对行进行切片。
- 通过标签选择，即 obj['b':'c']包含'c'行。
- 选择 row 和 column 的子集：ix。

Pandas 的功能非常强大，这里不再一一介绍，有兴趣的读者可以查阅官网的相关文档。图 6-1、图 6-2、图 6-3 和图 6-4 为笔者整理的 Pandas 相关使用的思维导图，供参考。



图6-1 Pandas的包导入、文件读取和对象创建

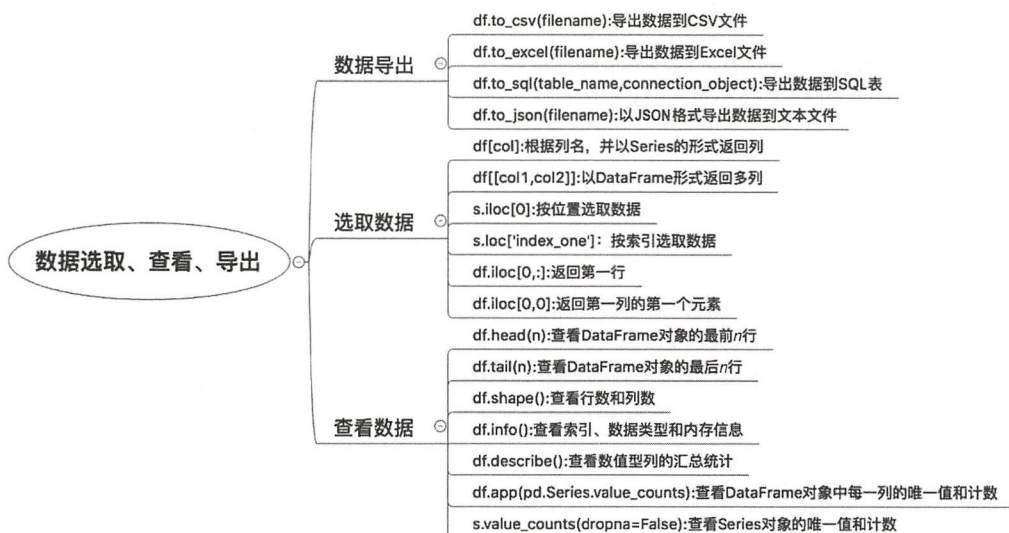


图6-2 Pandas的数据选取、查看和导出



图6-3 Pandas的数据统计和清洗



图6-4 Pandas的数据处理和合并

6.1.3 Python的优势和不足

Python 的优势不在于运行效率,而在于开发效率和高可维护性。对于数据处理的应用场景,从本质上说,我们把问题分解为两个方面。

(1) CPU 密集型操作。即我们要计算的大数据,大部分时间都在做一些数据计算,比如求逆矩阵、向量相似度、在内存中分词等,这种情况对语言的高效性非常依赖,Python 做此类工作时必然性能低下。

(2) I/O 密集型操作。假如大数据涉及频繁的 I/O 操作,比如从数据流中每次读取一行,然后不做什么复杂的计算,频繁地输入/输出到文件系统,由于这些操作调用的都是操作系统接口,所以用什么语言已经不重要了。

用 Python 来做整个流程的框架,然后核心的 CPU 密集型操作部分调用 C 函数,这样开发效率和性能都不错,缺点是对团队的要求更高,尤其是涉及 Python+C 的多线程操作时。

1. Python处理大数据的优势

- 开发速度快、高效、代码量较少。
- 丰富的扩展包,无论是正则表达式还是 HTML、XML 的解析,用起来都非常方便。
- 编码问题处理起来比较方便。

2. Python处理大数据的劣势

- 多线程问题。Python 线程有 gil,也就是说,Python 的多线程只能在一个核上运行,浪费了多核的优势。同时,当多个线程需要进行数据共享时,尤其是共享较大的数据时,会导致内存不足。
- Python 在处理大数据方面的执行效率不高。虽然类似 pypy(一个 JIT 的 Python 解释器)这样的工具能够在一定程度上提高数据处理速度,但是 pypy 的问题在于支持的 Python 经典的包有限,在使用上不够灵活和方便。

综上所述,Python 在处理大数据方面不一定是最优的选择,但是可以和其他语言并行使用以发挥其优势。如果处理的数据量不大(100MB 以下),Python 可能是一个不错的选择。

6.2 分布式离线计算

简单来说, 分布式计算是指把一个大计算任务拆分成多个小计算任务分布到若干机器上进行计算, 然后进行结果汇总。分布式计算的目的是可以计算海量的数据。对于海量计算, 最开始的方案是提高单机计算性能, 如采用大型主机, 由于数据的爆发式增长, 单机性能跟不上, 才有了分布式计算这种妥协方案。因为计算一旦拆分, 问题就会变得非常复杂, 像一致性、数据完整性、通信、容灾、任务调度等问题也都出现了。Hadoop 就是为了解决这些问题而诞生的。

Hadoop 是一个海量数据计算存储的基础平台架构, 它的最核心设计就是 HDFS 和 MapReduce, 其中 HDFS 为海量数据提供了存储能力, MapReduce 为海量数据提供了计算能力。

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台, 用户可以轻松地在 Hadoop 上开发和运行处理海量数据的应用程序。Hadoop 主要有以下几个优点。

- 高可靠性: Hadoop 按位存储和处理数据的能力值得信赖。
- 高扩展性: Hadoop 是在可用的计算机集群间分配数据并完成计算任务的, 这些集群可以方便地扩展到数以千计的节点上。
- 高效性: Hadoop 能够在节点之间动态地移动数据, 并保证各个节点的动态平衡, 因此处理速度非常快。
- 高容错性: Hadoop 能够自动保存数据的多个副本, 并且能够自动将失败的任务重新分配。
- 低成本: 与一体机、商用数据仓库等数据集市相比, Hadoop 是开源的, 因此项目的软件成本会大大降低。

6.2.1 MapReduce 离线计算

MapReduce^[3]是一种分布式计算模型, 由 Google 提出, 主要用于搜索领域, 解决海量数据的计算问题。

首先我们来了解 MapReduce On Yarn (Hadoop 通过 Yarn 来进行资源管理和任务调度) 的执行过程。如图 6-5 所示, 有三个大模块, 分别是 ResourceManager、NodeManager 和 MR ApplicationMaster (MR App Mstr)。

- ResourceManager：处理 Client 请求，启动或监控 MR ApplicationMaster，监控 NodeManager，调度和分配资源。
- NodeManager：单个节点上的资源管理，处理来自 ResourceManager 和 MR ApplicationMaster 的命令。
- MR ApplicationMaster：为应用程序申请资源，并分配给内部任务，切分数据，对任务进行监控和容错。

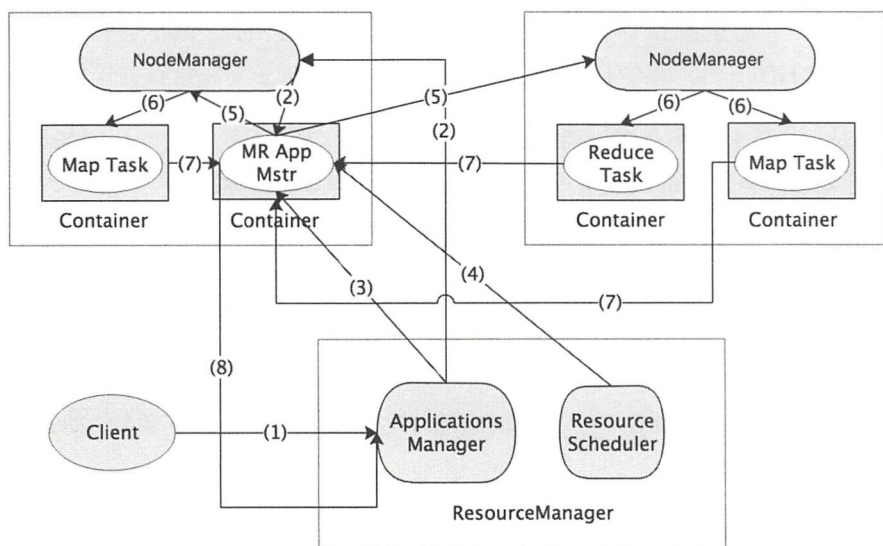


图6-5 MapReduce on Yarn的执行过程

MapReduce 的具体执行流程如下：

- (1) Client 向 ResourceManager 发出请求。
- (2) ResourceManager 接收到请求后，内部的 ApplicationsManager 会通知 NodeManager 创建 MR ApplicationMaster。
- (3) MR ApplicationMaster 创建完成之后，会通知 ResourceManager 自己已经创建完毕并注册，同时向 ResourceScheduler 申请资源。
- (4) ResourceScheduler 将资源分配给 MR ApplicationMaster。

(5) MR ApplicationMaster 通知 NodeManager 启动 Map Task 和 Reduce Task。

(6) NodeManager 启动 Map Task 和 Reduce Task 的 Container。

(7) Map Task 和 Reduce Task 将执行结果反馈给 MR ApplicationMaster。

(8) MR ApplicationMaster 将执行结果反馈给 ApplicationsMagager。

最后, 可以通过浏览器访问节点的 MR ApplicationMaster, 查看 Map Task 和 Reduce Task 的执行情况。

由上面的执行流程可知, MapReduce 在提交的过程中包含 Map Task 和 Reduce Task。

1. MapReduce执行原理

整个计算分成两个阶段: Map 和 Reduce。在 Map 阶段, 并行处理输入数据; 在 Reduce 阶段, 对 Map 结果进行汇总。Shuffle 连接 Map 和 Reduce 两个阶段, Map Task 将数据写到本地磁盘, Reduce Task 从每个 Map Task 上读取一份数据。用户只需要实现 map()和 reduce()两个函数, 即可实现分布式计算, 这两个函数的形参是 key-value 对, 表示函数的输入信息。MapReduce 的执行过程如图 6-6 所示。

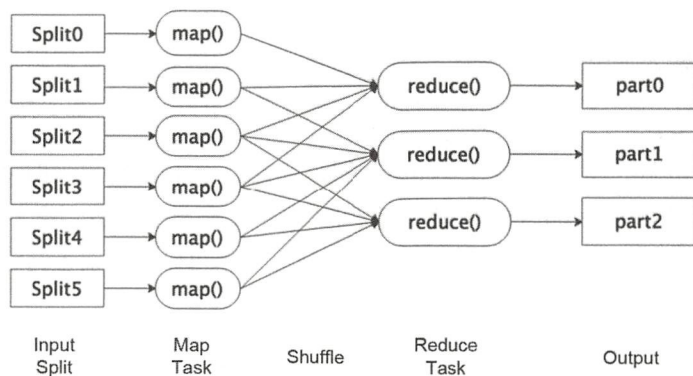


图6-6 MapReduce的执行过程

(1) Mapper 任务的执行过程

每个 Mapper 任务都是一个 Java 进程, 它会读取 HDFS 中的文件, 将文件解析成很多 key-value 对 (键值对) (在默认情况下, 行号作为 key, 内容作为 value), 经过 map()函数处理后, 转换为新的键值对再输出。整个 Mapper 任务的执行过程可以分为 6 个阶段, 如图 6-7 所示。

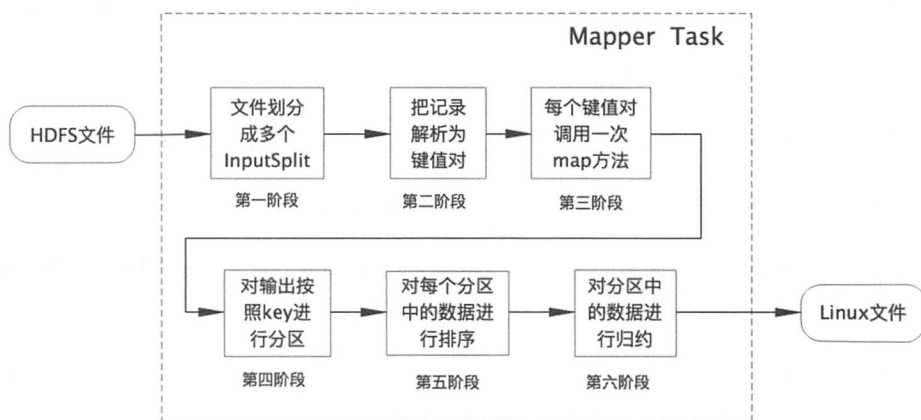


图6-7 Mapper任务的执行过程

第一阶段：把输入文件按照一定的标准分片，每个输入片的大小是不变的。在默认情况下，输入片的大小与数据块（Block）的大小是相同的。数据块的大小默认为 64MB。假设输入文件有两个，其中一个大小是 32MB，另一个大小是 82MB。小文件是一个输入片，大文件会分为两个输入片，那么这两个输入文件一共产生三个输入片。每一个输入片由一个 Mapper 进程处理，因此会有三个 Mapper 进程。

第二阶段：对输入片中的记录按照一定的规则解析成键值对。默认规则是把每一行的文本内容解析成键值对。“键”是每一行的起始位置（单位是字节），“值”是本行的文本内容。

第三阶段：开始调用 Mapper 类中的 map 函数。对第二阶段解析出来的每一个键值对，调用一次 map 函数。假如有 1000 个键值对，就会调用 1000 次 map 函数，每一次调用 map 函数时都会输出零个或者多个键值对。map 函数的具体执行逻辑可以由开发者通过重载默认的 map 函数来完成。

第四阶段：按照一定的规则对第三阶段的每个 Mapper 任务输出的键值对进行分区。分区是基于键进行的。比如键表示地域（如北京、上海、深圳等），那么就可以按照不同的地域进行分区，将同一个地域的键值对划分到一个分区中。默认只有一个分区，分区的数量就是 Reducer 任务运行的数量。

第五阶段：对每个分区中的键值对进行排序。首先，按照键进行排序，对于键相同的键值对，按照值进行排序。比如有三个键值对<2,4>、<1,2>、<2,5>，键和值分别是整数，那么排序后的结果是<1,2>、<2,4>、<2,5>。如果有第六阶段，那么进入第六阶段；如果没有，则直接输

出到本地文件中。

第六阶段：本阶段默认是没有的，需要开发者增加这一阶段的代码。在这个阶段对数据进行归约处理，也就是进行 Reduce 处理。只有键相等的键值对，才会调用一次 reduce() 函数。经过这一阶段，数据量会减小。将归约处理后的数据输出到本地的 Linux 文件中。

（2）Reducer 任务的执行过程

每个 Reducer 任务都是一个 Java 进程。Reducer 任务接收 Mapper 任务的输出，归约处理后写入 HDFS 中。整个 Reducer 任务的执行过程可以分为 3 个阶段，如图 6-8 所示。

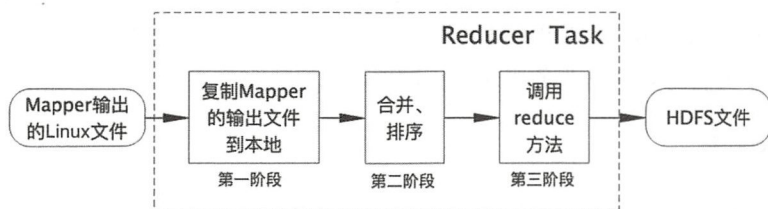


图6-8 Reducer任务的执行过程

第一阶段：Reducer 进程会主动从 Mapper 进程复制其输出的键值对。Mapper 进程可能会有很多，因此一个 Reducer 进程会复制多个 Mapper 进程的输出。

第二阶段：把 Reducer 复制的本地数据全部进行合并，即把分散的数据合并成一个大数据，然后再对合并后的数据排序。

第三阶段：对排序后的键值对调用 reduce() 函数。对于 key 相等的键值对调用一次 reduce() 函数，此时，开发者需要按业务需求对 reduce() 函数进行覆盖。调用覆盖后的方法会产生零个或者若干个键值对，最后把这些输出的键值对写入 HDFS 文件中。

2. Map和Reduce编程模型

在 Hadoop 中，map() 函数位于内置类 Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> 中，reduce() 函数位于内置类 Reduce<KEYIN, VALUEIN, KEYOUT, VALUEOUT> 中。我们编程要做的就是覆盖 map() 和 reduce() 函数。由 Hadoop 的 map() 和 reduce() 函数处理的数据是键值对，所以 map() 函数输入的数据是键值对，即两个参数，输出的结果也是键值对；reduce() 函数输入的参数和输出的结果也都是键值对。

（1）Mapper 类

在 Mapper 类中，有 4 个泛型，分别是 KEYIN、VALUEIN、KEYOUT 和 VALUEOUT，其

中 KEYIN、VALUEIN 指的是 map() 函数输入的参数 key 和 value 的类型；KEYOUT、VALUEOUT 指的是 map() 函数输出的 key 和 value 的类型。Map() 函数的定义如代码 6-1 所示。

代码 6-1

```
public void map(KEYIN key, VALUEIN value,
               Context context) throws IOException, InterruptedException {
    context.write((KEYOUT) key, (VALUEOUT) value);
}
```

在上面的代码中，输入的参数 key 和 value 的类型分别是 KEYIN、VALUEIN，每一个键值对都会调用一次 map() 函数。在这里，map() 函数没有处理输入的参数 key、value，而是直接通过 context.write() 方法输出的，输出的 key 和 value 的类型分别是 KEYOUT、VALUEOUT。这是默认实现，通常我们需要根据业务逻辑进行覆盖。

(2) Reducer 类

在 Reducer 类中，也有 4 个泛型，分别指的是 reduce() 函数输入的参数 key 和 value 类型，以及输出的 key 和 value 类型。下面看一下 reduce() 函数的定义，如代码 6-2 所示。

代码 6-2

```
public void reduce(KEYIN key, Iterable<VALUEIN> values, Context context
                  ) throws IOException, InterruptedException {
    for(VALUEIN value: values) {
        context.write((KEYOUT) key, (VALUEOUT) value);
    }
}
```

在上面的代码中，reduce() 函数的形参 key 和 value 的类型分别是 KEYIN、VALUEIN。需要注意的是，这里的 value 是存在于 java.lang.Iterable<VALUEIN> 中的，这是一个迭代器，用于集合遍历，这意味着 values 是一个集合。reduce() 函数的默认实现是把每个 value 和对应的 key 通过调用 context.write() 输出，这里输出的类型分别是 KEYOUT、VALUEOUT。通常我们会根据业务逻辑覆盖 reduce() 函数的实现。

3. 单词计数举例

该业务要求统计指定文件中所有单词出现的次数。

思路：通过前面介绍我们了解到，在 Mapper 任务的执行过程中，第二阶段是把文件的每一行转换成键值对，第三阶段是调用 map() 函数获取每一行文本内容。因此，可以在 map() 函数中计算本行文本中单词出现的次数，并把每个单词出现的次数作为新的键值对输出。在 Reducer

任务的执行过程中，第二阶段会对 Mapper 任务输出的键值对按照键进行排序，对于键相等的键值对会调用一次 reduce() 函数。在这个例子中，“键”是单词，“值”就是单词出现的次数。因此，可以在 reduce() 函数中对单词在不同行中出现的次数进行累加，结果就是该单词出现的总次数，最后输出这个结果。

以下代码分两部分，其中代码 6-3 主要包括 map() 和 reduce() 函数，代码 6-4 是启动类。

代码 6-3

```
package com.demo.bigdata.mr;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

    public static class WordCountMap extends
        Mapper<LongWritable, Text, Text, IntWritable> {

        private final IntWritable one = new IntWritable(1); // 表示单词在该行中出现的次数
        private Text word = new Text(); // 表示该行中某一单词

        public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer token = new StringTokenizer(line);
            while (token.hasMoreTokens()) {
                word.set(token.nextToken());
            }
        }
    }
}
```

```

        context.write(word, one);
    }
}

public static class WordCountReduce extends
    Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
        int sum = 0; // 表示单词出现的总次数
        for (IntWritable val : values) {
            sum += val.get(); // 执行到这里，sum 表示该单词出现的总次数
        }
        context.write(key, new IntWritable(sum)); // key 表示单词，是最后输出的 key，sum
        // 表示单词出现的总次数，是最后输出的 value
    }
}

```

代码 6-4

```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf); // 创建一个 job 对象，封装运行时所需要的所有信息
    job.setJarByClass(WordCount.class);
    job.setJobName("wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(WordCountMap.class); // 设置自定义的 Mapper 类
    job.setReducerClass(WordCountReduce.class); // 设置自定义的 Reducer 类

    job.setInputFormatClass(TextInputFormat.class); // 设置把输入文件处理成键值对的类
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0])); // 告诉 job 执行作业时的输入路径
    FileOutputFormat.setOutputPath(job, new Path(args[1])); // 告诉 job 执行作业时的输出
    // 路径

    job.waitForCompletion(true); // 让作业运行，直到运行结束，程序退出
}

```


在代码 6.3 和代码 6.4 中 WordCountMap 类的泛型依次是 LongWritable, Text, Text, IntWritable。map() 函数的第二个形参是行文本内容，这是我们所关心的。核心代码表示把行文本内容按照空格进行拆分，把每个单词作为新的键，数值 1 作为新的值，写入上下文 context 中。在这里，因为输出的是每个单词，所以其出现次数是常量 1。如果一行文本中包含两个 hello，则会输出两次 <hello, 1>。

在以上代码中，我们创建了一个 job 对象，这个对象封装了任务，可以提交给 Hadoop 独立运行。最后一句 job.waitForCompletion(true)，表示把 job 对象提交给 Hadoop 运行，直到作业运行结束。

6.2.2 离线计算的数据倾斜问题

数据倾斜就是指我们在计算数据的时候，数据的分散度不够，导致大量的数据集中到一台或者几台机器上，这些数据的计算速度远远低于平均计算速度，导致整个计算过程很慢。

Hadoop 中的数据倾斜主要表现在任务在 Reducer 阶段会长时间停留在大概 99% 处不能结束。这时如果仔细查看日志就会发现有一个或多个 Reducer 执行过程报 OOM (Out Of Memory) 错误或者 Container 加载失败，这时基本可以判断本次离线计算任务的输入数据可能存在数据倾斜问题。

导致数据倾斜的原因可能有很多种，常见的原因如下。

- 某些 SQL 语句本身有问题。比如 SQL 中含有 distinct 的计算。
- key 分布不均匀。比如在两个表 join 过程中，其中一个表的 key 分布不均匀。
- 业务数据本身有问题。比如业务数据 join 的 id 分布不均匀，或者 id 为 null 值的比重较大，都会引起数据倾斜。
- 建表时考虑不周。比如同为 type 字段，其中一个表类型为 int，另一个表类型为 string。

下面列举了一些常见的导致数据倾斜的场景。

场景 1：当一个大表和一个表 join 时，如果小表的 key 较集中，将会引起大表中的数据被分发到一个或者少数几个 Reducer 任务中，导致数据分布不均匀。

场景 2：在 group by 时，如果分组的维度太少，维度的值分布不均匀，将导致数据分布不均匀。

场景 3：当大表与大表关联时，在关联的条件字段中，其中一个表的空值、null 值过多，将导致数据分布不均匀。

针对数据倾斜，业界一般有以下几种解决方案。

1. 调节参数

可以通过修改 `hive.map.aggr` 和 `hive.groupby.skewindata` 参数同时配置为 `true`，在 Mapper 端进行聚合操作，当发生数据倾斜时进行负载均衡。所生成的查询计划会有两个 MR 任务。在第一个 MR 任务中，Mapper 阶段的输出结果集合会被随机分布到 Reducer 阶段中，每个 Reducer 都进行部分聚合操作，并输出结果。这样处理的结果是相同的 Key 可以被分发到不同的 Reducer 中，从而达到负载均衡的目的。在第二个 MR 任务中，Mapper 根据第一个 MR 任务预处理后的数据结果再按照 key 输出给 Reducer，这个过程可以保证相同的 key 被分布到同一个 Reducer 中。经过这两轮 MR 任务最后完成最终的聚合操作。相关的参数设置如下：

```
hive.map.aggr=true
hive.groupby.skewindata=true
```

2. 优化SQL语句

- 使用 `mapjoin`：让小的维度表（建议在 20000 条记录以下）先写入内存，并按顺序扫描大表完成 join。这种方式比较适用于大表和小表的 join。
- 空值优化：可以将空值的 key 变成一个字符串加上随机数，把倾斜的数据分布到不同的 Reducer 中。也可以对空值进行单独处理，然后再和其他非空值的计算结果进行合并。
- `group by` 优化：采用 `sum()` 结合 `group by` 的方式替换 `count(distinct)` 来完成计算。

3. 特殊情况特殊处理

在业务逻辑优化效果不太好的情况下，有些时候可以将倾斜的数据单独拿出来处理，最后再进行 union。为了方便理解，下面列举几个业务场景来进行说明。

案例 1：空值产生的数据倾斜问题。

场景：比如在日志中，通常会发生信息丢失的问题。假如日志中的 `order_id` 存在丢失情况，如果将其中的 `order_id` 和订单表的 `order_id` 关联，就会出现数据倾斜。

解决方法 1：`order_id` 为空值的则不参与关联，用 `union all` 合并数据，如代码 6-5 所示。

代码 6-5

```
select * from log a
  join orders b
on a.order_id = b.order_id
  and a.order_id is not null
union all
select * from log a
  where a.order_id is null;
```

解决方法 2：为空值分配一个随机值，如代码 6-6 所示。

代码 6-6

```
select *
  from log a
 left outer join orders b
on case when a.order_id is null then cast(rand()*10000 as bigint) else a.order_id end
= b.order_id;
```

总结：解决方法 2 比方法 1 的执行效率更高，不但 I/O 少了，而且作业数也少了。在解决方法 1 中 log 读取两次，job 数是 2；在解决方法 2 中 job 数是 1。这种优化适合由于无效 id（比如 -99、”、null 等无效字符组合）产生的倾斜问题。把空值的 key 变成一个字符串加上随机数，就能把倾斜的数据分布到不同的 Reducer 中，从而解决数据倾斜问题。

案例 2：小表不小不大，怎么用 mapjoin 解决数据倾斜问题。

使用 mapjoin 解决小表（记录数少）关联大表的数据倾斜问题。这种方法使用的频率非常高，但是如果小表很大，大到 mapjoin 会出现 bug 或异常，这时就需要特别处理了。例如：

代码 6-7

```
select /*+mapjoin(b)*/ from log a
 left outer join orders b
on a.order_id = b.order_id;
```

orders 表有超过 600 万条的记录，把 orders 分发到所有的 Mapper 中也有不小的开销，而且 mapjoin 不支持这么大的表。如果用普通的 join，又会碰到数据倾斜的问题。那么解决方法如代码 6-8 所示。

代码 6-8

```
select /*+mapjoin(t)*/ from log a
 left outer join (
  select /*+mapjoin(c)*/ b*
```



```

from ( select order_id from log group by order_id) c
join orders b
on c.order_id = d.order_id
) t
on a.order_id = t. order_id

```

综上所述，解决数据倾斜问题就是要将 Mapper 阶段的输出数据更均匀地分布到 Reducer 中，可以通过改变 job 的步骤、处理 key 值等方式来实现。数据倾斜多数是由于开发人员疏忽引起的，问题本身并不是很复杂。

6.2.3 分布式离线计算的技术栈

表 6-1 中列出了分布式离线计算知识点分类及相关技术栈。

表 6-1 分布式离线计算知识点分类及相关技术栈

知识点分类	相关技术栈
网络通信类（网络是分布式的基础）	OSI 模型的 7 层、TCP/IP、DNS、NAT、HTTP、SPDY/HTTP2、Telnet
网络编程（通过程序实现在多台主机之间进行通信）	Socket、多线程、非阻塞 I/O、网络框架、Netty、Mina、MQ、同步 RPC、异步 RPC，以及一些主要的 RPC 协议，比如 RMI、Rest API 和 Thrift
集群类（将 N 台主机当成一个系统）	<ul style="list-style-type: none"> 高可用性：保证服务一直处于可用状态，如采用冗余的设备、多副本 负载均衡：将大量的工作负载均匀分配到多台主机上 伸缩性（横向）：技术保障可通过添加计算机或设备来解决业务增长带来的压力 分片：把数据分成多个数据集，由不同的服务器来分别处理 容错性：当硬件或软件发生故障时能够继续运转 故障检测：心跳包、告警、性能预警 故障转移：当出现错误时自动解决 分布式一致性：强一致性、最终一致性 集群状态协调：如 Zookeeper、分布式锁、选主 一致性哈希：将数据分布到集群中的多台主机上 分布式事务：保证在多台服务器上完成的操作符合事务的 ACID 属性
安全类（网络需要基础的安全保障）	<ul style="list-style-type: none"> 身份认证：基于用户名/口令、基于数字证书 私密性：对称加密、非对称加密 完整性：安全散列、消息认证码（MAC） 不可否认性：基于数字证书的数字签名和验签、基于密钥的散列



6.3 案例分析

离线数据为多链路上下游监控提供了更加精准的数据支持,是监控体系的很重要的一部分。本节将给大家介绍微博广告的离线计算。微博广告系统每天会产生 10TB 以上的增量数据。微博有超级粉丝通、粉丝经济等丰富的广告产品,由于微博具有庞大的用户关系和复杂的业务场景,数据仓库应该如何选择数据模型,如何在满足业务需求的同时能高效、准确地按时完成计算,都存在巨大的挑战。离线计算,主要需求是计算历史数据,关于当天的批处理或流式计算不在本节内容范围内。微博广告数据架构示意图如图 6-9 所示。

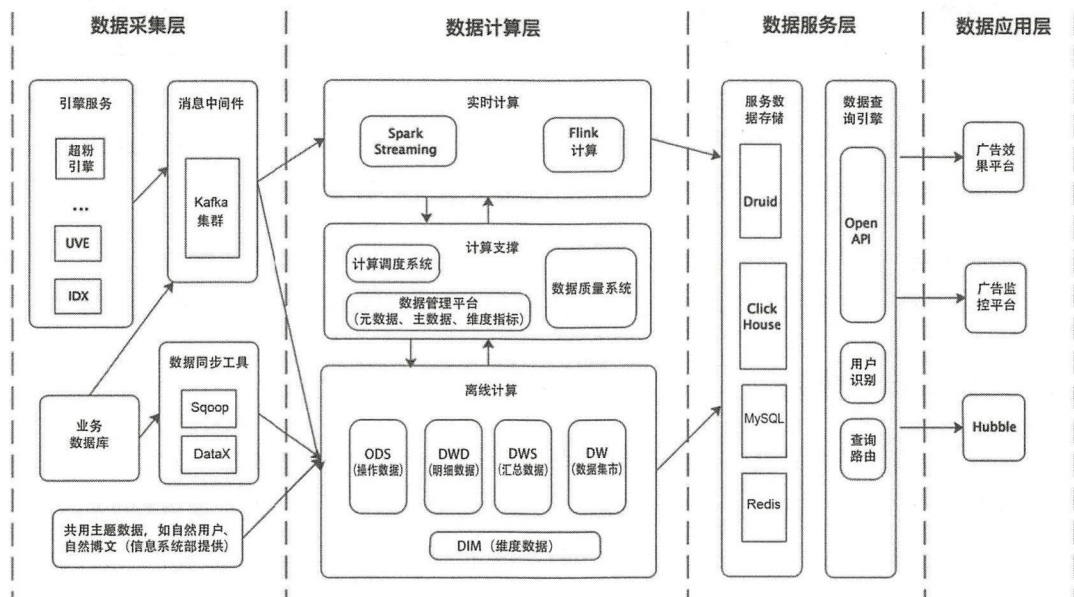


图6-9 微博广告数据架构示意图

微博广告数据架构主要分为四层,分别是数据采集层、数据计算层、数据服务层和数据应用层。

数据采集层:数据采集是大数据架构最重要的一环。微博广告的数据采集主要分三大块,第一块是数据同步工具,从业务数据库同步数据,可支持增量和全量操作;第二块是日志采集技术,在 Web、H5 端采用通用的 JS 模块,在移动客户端采用 SDK 捕获各种事件,当 JS 模块和 SDK 拿到事件后,转换成消息或落地成文件;第三块是消息中间件,接收第二块的消息,使用 Kafka 处理,也会用 Flume 来检测落地的文件,然后再转换成消息。

图 6-9



数据计算层：首先是实时计算部分，Spark Streaming、Storm、Flink 等工具直接从消息中间件消费消息，然后计算处理，将处理结果发送到数据服务层的数据存储模块。其次是离线计算部分，微博广告的数据离线计算量非常大，可以满足日常业务支撑的大部分数据需求。离线计算按照业界的标准分层处理，主要有操作数据层、明细数据层、汇总数据层、数据集市层。分层之后的计算脉络更清晰，非常有利于对后续计算的维护。最后是对元数据、主数据、计算调度的管理。这是整个计算层的核心支撑，非常重要，包含了数据质量系统。元数据是指管理数据的数据，比如表的表、字段的表等。主数据是指能在各个系统之间共享的数据，比如地域字典、平台字典等。

数据服务层：主要对外提供数据 API 服务，主要分为数据服务工具（又称查询引擎）和服务层的数据存储（服务数据源）。查询引擎有两个职能，一是提供 OpenAPI 的 Web 服务；二是对查询操作进行路由，保障查询的性能。查询引擎一定是可扩展的。服务层的数据存储比较重要，需要支持复杂的海量数据查询，同时以毫秒级响应。微博广告的选型为 Druid 和 ClickHouse 的组合，其中 Druid 主要计算实时数据，ClickHouse 主要支持离线数据。

数据应用层：专为数据产品而设计，如广告的投放监控、定投产品等。数据按需加工处理后，可以满足某种或多种数据产品的需求。数据应用层是从数据服务层提交的 API 获取数据的。

6.3.1 离线计算管理

业务复杂度高，离线计算的数据种类就多，由“多”发展下去可能就是“乱”，由“乱”发展下去必然是差。为解决多、乱、差的问题，引入了数据仓库的概念。我们认为数据仓库是一种对数据进行管理和使用的方式，它是包括 ETL、调度、建模在内的一套理论体系。也就是说，数据仓库也是离线计算的管理方式，管理离线计算是数据仓库的一个重要功能。

1. ETL

微博广告计算有一套标准的 ETL 代码模板，能兼容开发环境和线上环境。我们的经验是 ETL 只需要关注业务逻辑，将打印日志、计算状态监控都交给调度系统，这些逻辑的代码不应该放在 ETL 中进行处理，否则会增加 ETL 维护的复杂度。

2. 调度

有一个好用的调度系统，离线计算管理就成功了一大半。调度系统又称作工作流系统，开源的有 Oozie、Airflow、Azkaban 等。值得一提的是，一个调度系统至少需要满足如下几点：



①支持有向无环图，即支持计算节点的单项依赖或多重依赖配置，数据流向一致。②支持计算节点的自动重算和超时配置。这两项功能非常实用，特别是当调度任务比较多时，能极大提高任务管理和调度的效率。③具有失败报警功能。经过调研，微博广告使用了新浪平台团队自己开发的调度系统，定制了很多好用的功能，比如以项目方式管理计算原子、友好的可视化界面、灵活的报警方式等。

6.3.2 离线计算原子控制

离线计算原子，通常指一个 ETL 的计算单元。离线计算原子究竟要控制多少计算逻辑，以及控制度如何，在进行数据架构时容易被忽视。如果在这方面没有形成规范和共识，结果可能会导致每个人都按照自己的经验来控制处理逻辑，从而导致有些计算单元特别耗时，有的 MR 任务的实际计算时间（map 任务从 0% 开始到 100% 所需要的时间）比 MR 任务提交的时间（主要包含资源调度、队列等待等时间，在 MR 任务实际计算之前）还短。整个计算脉络节奏不一，维护起来比较困难。我们总结的经验是：

- 按照业务系统的事务逻辑控制，即逻辑的所有步骤要么都成功，要么都失败。
- 参考计算时间，输入文件较大且有较多的计算时，要拆分成多个计算单元，用空间换时间。
- 计算原子一定要可重复计算，即在计算完成输出时一定要覆盖原来的文件，或者在计算前就直接删除原来的计算结果文件。

上面第三点看起来很简单，但是在实践过程中还是容易犯这类错误和。比如使用 MySQL 的 insert 方法将数据插入到 MySQL 数据库中，原来的数据没有清除，导致重新计算后量级翻倍；再比如，在 Hive 里使用 insert into 方法，导致局部数据重复。

6.3.3 离线计算的数据质量

数据的准确性基本上是离线计算的生命线，如果数据不准确，计算就没了意义。那么如何校验所产出的数据的质量呢？微博广告数据团队花大功夫开发了一套数据质量管理体系，该系统主要从数据的完整性、一致性、准确性、及时性四个角度进行了校验。完整性是指日志或抽取的数据条数与实际的数据的比值，理论值为 100%。例如，每天的曝光日志至少为 1.5TB，如果某一天曝光日志只有 1TB，则数据缺少了 33%，这就说明数据不完整。准确性是指数值的准



确性。比如订单金额字段是不可能为负值的，如果为负值，则说明不满足准确性。一致性是指同一个指标在同一口径下，对于不同的数据源或不同的表计算，量级是一样的。及时性表现在数据按时计算完成上，如果没有按时计算完成，数据的价值就会大打折扣。微博广告的数据质量系统架构示意图如图 6-10 所示。

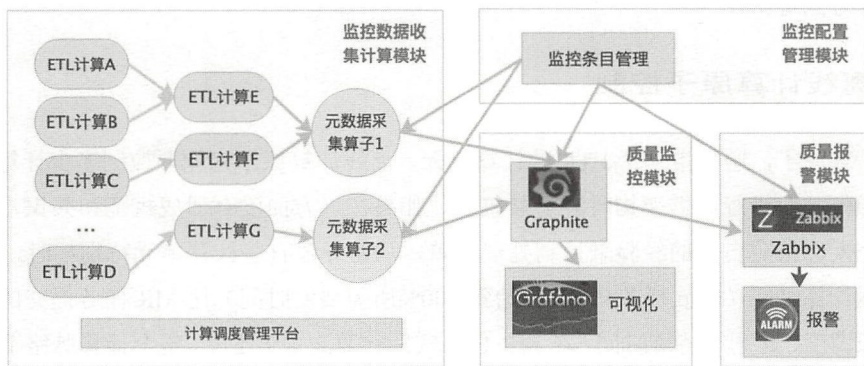


图6-10 数据质量系统架构示意图

- 监控配置管理模块：添加管理监控的条目，配置报警阈值，删除或修改监控条目。
- 监控数据收集计算模块：跨表采集或单表采集数据的记录数、字段长度数、空值、主键重复、枚举值、连续值等监控数据，并将数据存储在 HDFS 上。
- 质量监控模块：数据采集算子将所采集的数据存入 Graphite，数据用 Grafana 可视化展示出来。
- 质量报警模块：启动 Zabbix 的 Trigger 模块，根据报警阈值比较数据，触发报警。

数据质量系统是微博广告数据仓库的一把利剑，它有效地解决了数据仓库中数据质量的问题。

6.4 本章小结

本章重点介绍了 MapReduce 在离线计算上的架构设计原理、执行流程和编程模型，同时还简单介绍了在离线计算方面如何处理数据倾斜问题。本章最后通过微博广告的大数据整体架构，介绍了离线计算的相关技术。



6.5 参考文献

- [1] Dale Dougherty, Arnold Robbins 著. 张旭东, 杨作梅, 田丽华等译. Sed 与 awk. 北京: 机械工业出版社, 2003 年 6 月
- [2] <http://pandas.pydata.org/pandas-docs/stable/10min.html>
- [3] <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>
- [4] https://www.chrisstucchio.com/blog/2013/hadoop_hatred.html
- [5] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

第7章

实时计算框架

实时流计算就是近几年由于数据被广泛重视，在金融、广告、网络监控等行业需要实时地从大量的数据中获取可用的信息，通过实时推荐及计算来获取目标数据而兴起的技术。流式计算处理的业务特点是数据的价值随着时间的流逝而降低，所以提高数据的处理速度及实时性是极其重要的。例如，用户在浏览微博时插入了 Feed 广告，我们需要对所插入广告的曝光、互动、负反馈等信息进行及时的反馈，这时就需要流式计算。

针对海量的数据，要做到具有很高的实时性。但是对于传统的数据，直接录入几十亿、上百亿级数据是不太现实的。对于早先使用 Hadoop MR 通过配置调度计算离线小批量的数据是没有办法达到实时流的实时性要求的。目前大型互联网公司的流量都是海量的，而且数据流的流量不是恒定的，如微博的热门事件会突然让流量翻倍，这些流量是没法预估的。因此，这对实时流处理的横向扩展及吞吐量提出了很高的要求。

7.1 关于实时流计算

7.1.1 如何提高实时流计算的实时性

实时流计算数据一般是通过消息中间件源源不断地传输到计算引擎的。对于计算引擎来说，例如 Spark Streaming，在特定的时间窗口进行计算。这个时间窗口可以提高数据的吞吐量，减小数据在各个分布式节点发送的频率；弊端是数据实时性降低了。实时流计算对于实时流来说是基础。

7.1.2 如何提高实时流计算结果的准确性

实时流计算结果的准确性是整个实时流计算的核心。那么如何处理数据才能让计算结果更准确呢？实时流有三种时间选择：数据进入计算引擎的时间、数据产生的时间和日志的时间。时间选择对数据计算结果的准确性有很大的影响。当然，实时流计算结果的准确性对于业务理解也是相当重要的。抛开业务，集群故障等容错能力对于计算也是很重要的。

7.1.3 如何提高实时流计算结果的响应速度

实时流计算结果如何快速响应、计算结果是时序值还是聚合值、大量的秒级时序值如何存储，都是实时流计算需要考虑的，存储的吞吐量也需要评估。存储写入吞吐量和查询响应速度对于实时流计算结果的响应是至关重要的。

7.2 Spark Streaming 计算框架介绍

7.2.1 概述

Spark Streaming^[1]是 Spark 核心 API 的一个扩展，可以实现高吞吐量、具有容错机制的实时流处理。可以从 Kafka、Socket、Flume 数据源获取数据，然后对数据采用 map、reduce、join 和 window 等高级函数进行复杂的计算处理，最后将计算结果保存到文件系统、数据库等存储中。还可以进行机器学习、图计算等数据处理。Spark Streaming 处理数据流图如图 7-1 所示。



图7-1 Spark Streaming 处理数据流图（图片来源于Spark官网^[1]）

Spark Streaming 提供了一个高层抽象，称为离散流或 DStream，它表示连续的数据流。DStream 可以通过 Kafka、Flume 和 Kinesis 等来源的输入数据流创建，也可以通过在其他 DStream 上应用高级操作来创建。在内部，DStream 表示为一系列 RDD。Spark Streaming 数据处理过程如图 7-2 所示。

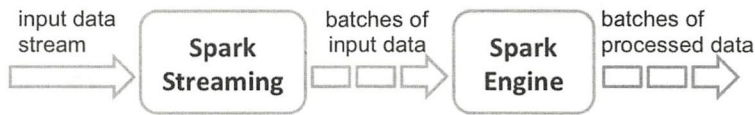


图7-2 Spark Streaming数据处理过程（图片来源于Spark官网^[1]）

RDD（弹性分布式数据集）是分布式内存的一个抽象概念。RDD 提供了一种高度受限的共享内存模型，即 RDD 是只读的记录分区的集合，只能通过在其他 RDD 中执行确定的转换操作（如 Map、Join 和 Group By）而创建。然而，这些限制使得实现容错的开销很低。对于开发者而言，RDD 可以看作是 Spark 的一个对象，它本身运行于内存中，如读文件是一个 RDD，对文件计算是一个 RDD，结果集也是一个 RDD，不同的分片、数据之间的依赖、key-value 类型的 map 数据都可以看作是 RDD。

7.2.2 基本概念

StreamingContext: StreamingContext 对象，它是所有 Spark Streaming 功能的主要入口点，要初始化 Spark Streaming 程序，必须创建一个 StreamingContext 对象。

DStream: Spark Streaming 抽象的数据集，代表源源不断的数据流。数据流通过数据源产生或者通过计算转化后的数据流得到。DStream 由一段连续的分布式弹性数据集组成，这里被称作 RDD。

batch data: Spark Streaming 并非严格意义上的实时流，属于微批处理，它是按照一定的时间切片将微小的批处理数据连续起来的实时数据。

batch interval: 时间片的大小，每一个时间片就是一组 RDD，对 RDD 只能进行转换操作。时间片是在 StreamingContext 中进行设置的。

window length: 时间窗口的长度，它必须是时间片的整数倍。

7.2.3 运行原理

Spark Streaming 是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如 Kafka、Flume 和 TCP 套接字）进行类似 Map、Reduce 和 Join 等复杂的操作，并将结果保存到外部文件系统或数据库中。

计算流程：①构建 Spark Application 的运行环境（启动 SparkContext），SparkContext 向资源管理器（可以是 Standalone、Mesos 或 YARN）注册并申请运行 Executor 资源。②资源管理器分配 Executor 资源并启动 StandaloneExecutorBackend，Executor 运行情况将随着心跳发送到资源管理器上。③SparkContext 构建有向无环图，将有向无环图分解成 Stage，并把 Taskset 发送给 TaskScheduler。Executor 向 SparkContext 申请 Task，TaskScheduler 将 Task 发放给 Executor 运行，同时 SparkContext 将应用程序代码发放给 Executor。④Task 在 Executor 上运行，运行完毕释放所有资源。

容错性：对于流式计算来说，容错性至关重要。首先我们要明确一下 Spark 中 RDD 的容错机制。每一个 RDD 都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系，所以只要输入数据是可容错的，那么当任意一个 RDD 的分区出错或不可用时，就都可以利用原始输入数据通过转换操作而重新算出。

实时性：由于 Spark Streaming 本身的架构设计是采用微批处理方式来处理数据的，所有数据都需要流经有向无环图，最小的 batch time 在秒级，所以对于对实时性要求较高的业务场景来说，Spark Streaming 不是很好的选择。

吞吐量：Spark 本身就是分布式计算引擎，可以很容易地增加节点和设置 Executor、Driver 的并行度。并且可以对计算结果进行缓存，极大地减小了不必要的资源消耗，提高了数据处理的吞吐量。

7.2.4 编程模型

DStream（Discretized Stream）作为 Spark Streaming 的基础抽象，它代表持续性的数据流。这些数据流既可以通过外部输入源来获得，也可以通过现有的 DStream 的转换操作来获得。在内部实现上，DStream 由一组在时间序列上连续的 RDD 表示，每一个 RDD 都包含了自己在特定时间间隔内的数据流，如图 7-3 所示。

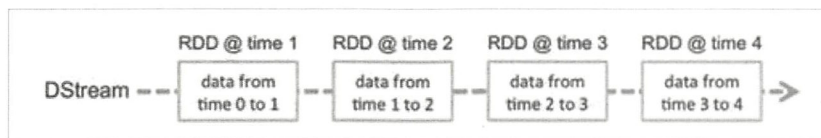


图7-3 DStream操作流

在 DStream 中在时间轴下生成离散 RDD 序列，如图 7-4 所示。

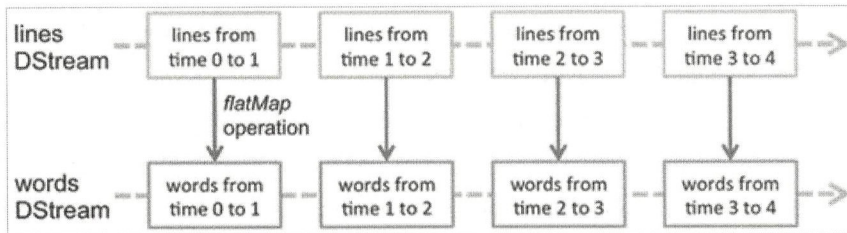


图7-4 DStream生成RDD序列的过程

对 DStream 中数据的各种操作也是映射到内部的 RDD 上来进行的，通过 RDD 的转换操作可以生成新的 DStream。这里的执行引擎是 Spark。

7.2.5 Spark Streaming 的使用

Spark Streaming 承袭了 Spark 的编程风格，对于已经了解 Spark 的用户来说能够快速上手。接下来，以 Spark Streaming 官方提供的 WordCount 代码为例来介绍 Spark Streaming 的使用方式，如代码 7-1 所示。

代码 7-1

```
import org.apache.spark.*;
import org.apache.spark.api.java.function.*;
import org.apache.spark.streaming.*;
import org.apache.spark.streaming.api.java.*;
import scala.Tuple2;

//Create a local StreamingContext with two working thread and batch interval of 1 second
SparkConf conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount");
JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(1));
JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
//Split each line into words
JavaDStream<String> words = lines.flatMap(x -> Arrays.asList(x.split(" ")).iterator());
//Count each word in each batch
JavaPairDStream<String, Integer> pairs = words.mapToPair(s -> new Tuple2<>(s, 1));
JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey((i1, i2) -> i1 + i2);

//Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print();
jssc.start();           // Start the computation
jssc.awaitTermination(); // Wait for the computation to terminate
```

(1) 创建 `JavaStreamingContext`。`JavaStreamingContext` 是 Spark Streaming 上下文，可以指定 Master 名称、批处理时间、运行模式，以及通过广播进行数据共享。

(2) 创建 `inputStream`。Spark Streaming 需要指明数据源，如上例中所示的 `socketTextStream`，Spark Streaming 以 Socket 连接作为数据源读取数据。当然，Spark Streaming 支持多种不同的数据源，包括 Kafka、Flume 等数据源。

(3) 转换 `DStream`。开发人员可以在 `DStream` 的基础上进行所需的操作，以 `WordCount` 为例，先对 `txt` 进行 `flatMap` 操作拆分单词，然后执行 `mapToPair` 操作将单词进行映射，接下来通过 `reduceByKey` 对相同的单词进行累加。

(4) 触发动作。上面的 `WordCount` 例子是进行单词统计的，然后打印结果数据（调用 `print()` 函数）。Spark Streaming 只有在触发动作时才会开始进行转换。

(5) 启动实时流。设置好 Spark Streaming 上下文、输入流，以及设计好整个计算的有向无环图后，我们就可以通过 `start()` 函数进行计算了。

与 RDD 类似，转换允许修改输入 `DStream` 中的数据。`DStream` 支持 Spark RDD 的许多转换操作。一些常见的转换方法如表 7-1 所示。

表 7-1 `DStream` 支持的常见的转换方法

方法	描述
<code>map(func)</code>	通过将源 <code>DStream</code> 的每个元素传递给函数 <code>func</code> 来返回一个新的 <code>DStream</code>
<code>flatMap(func)</code>	与 <code>map</code> 类似，但每个输入项目可以映射到 0 个或多个输出项目上
<code>filter(func)</code>	过滤筛选函数，通过仅选择 <code>func</code> 返回 <code>true</code> 的源 <code>DStream</code> 的记录来返回新的 <code>DStream</code>
<code>repartition(numPartitions)</code>	通过创建更多或更少的分区来更改 <code>DStream</code> 中的并行性级别
<code>union(otherStream)</code>	通过计算源 <code>DStream</code> 的每个 RDD 中元素的数量来返回一个新的单元 RDD 的 <code>DStream</code>
<code>count()</code>	通过计算源 <code>DStream</code> 的每个 RDD 中元素的数量来返回一个新的单元 RDD 的 <code>DStream</code>
<code>reduce(func)</code>	通过使用函数 <code>func</code> （它接收两个参数并返回一个）来聚合源 <code>DStream</code> 的每个 RDD 中的元素，从而返回一个新的单元 RDD 的 <code>DStream</code>
<code>countByKey()</code>	当在类型为 <code>K</code> 的元素的 <code>DStream</code> 上调用时，返回一个新的 <code>(K,Long)</code> 对的 <code>DStream</code> ，其中每个键的值都是其在源 <code>DStream</code> 的每个 RDD 中的频率

续表

方法	描述
<code>reduceByKey(func, [numTasks])</code>	当在(K,V)对的 DStream 上调用时, 返回一个新的(K,V)对的 DStream, 其中每个键的值都使用给定的 <code>reduce</code> 函数进行聚合。注意: 在默认情况下, 它使用 Spark 的默认并行任务数 (2 表示本地模式, 而在集群模式下, 该值由 <code>config</code> 属性 <code>spark.default.parallelism</code> 决定) 进行分组。你可以传递一个可选的 <code>numTasks</code> 参数来设置不同数量的任务
<code>join(otherStream, [numTasks])</code>	当在(K,V)和(K,W)对的两个 DStream 上调用时, 返回一个新的(K,(V,W))对的 DStream 与每个键的所有元素对
<code>cogroup(otherStream, [numTasks])</code>	当调用(K,V)和(K,W)对的 DStream 时, 返回一个新的(K,Seq[V],Seq[W])元组 DStream
<code>transform(func)</code>	通过对源 DStream 的每个 RDD 应用 RDD-RDD 函数来返回一个新的 DStream。这可以用来在 DStream 上执行任意的 RDD 操作
<code>updateStateByKey(func)</code>	返回一个新的“状态”DStream, 其中每个键的状态都通过对键的先前状态和键的新值应用给定函数来更新。这可以用来维护每个键的任意状态数据

Spark Streaming 还提供了窗口计算, 允许通过滑动窗口对数据进行转换。Spark Streaming 支持的窗口转换方法如表 7-2 所示。

表 7-2 Spark Streaming 支持的窗口转换方法

方法	描述
<code>window(windowLength, slideInterval)</code>	返回一个新的 DStream, 它是根据源 DStream 的窗口批次计算得出的
<code>countByWindow(windowLength, slideInterval)</code>	返回流中元素的滑动窗口计数
<code>reduceByWindow(func, windowLength, slideInterval)</code>	通过使用 <code>func</code> 在滑动窗口间隔中聚合流中的元素来创建新的单元素流
<code>reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])</code>	在(K,V)对的 DStream 上调用时, 返回一个新的(K,V)对, 其中每个键的值在滑动窗口中使用给定的 <code>reduce</code> 函数 <code>func</code> 进行批量聚合。注意: 在默认情况下, 它使用 Spark 的默认并行任务数 (2 表示本地模式, 而在集群模式下, 该值由 <code>config</code> 属性 <code>spark.default.parallelism</code> 决定) 进行分组。你可以传递一个可选的 <code>numTasks</code> 参数来设置不同数量的任务

续表

方法	描述
<code>reduceByKeyAndWindow(func,invFunc>windowLength,slideInterval,[numTasks])</code>	上述 <code>reduceByKeyAndWindow()</code> 的更高效版本。其中每个窗口的减少值都是使用前一个窗口的减少值递增计算的。这是通过减少进入滑动窗口的新数据并“反向减少”离开窗口的旧数据来完成的。当窗口滑动时，“添加”和“减去”键的计数就是一个例子。但是，它仅适用于“可逆减少函数”，即那些具有相应“反向减少”函数的函数（作为参数 <code>invFunc</code> ）。与 <code>reduceByKeyAndWindow</code> 一样， <code>reduce</code> 任务的数量可通过可选参数进行配置。请注意，必须启用检查点才能进行此操作
<code>countByValueAndWindow(windowLength,slideInterval,[numTasks])</code>	在 (K,V) 对的 <code>DStream</code> 上调用时，返回一个新的 $(K,Long)$ 对的 <code>DStream</code> ，其中每个键的值都是滑动窗口内的频率。与 <code>reduceByKeyAndWindow</code> 一样， <code>reduce</code> 任务的数量可通过可选参数进行配置

Spark Streaming 窗口计算示意图如图 7-5 所示。

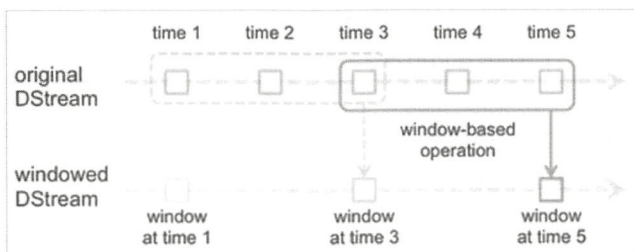


图7-5 Spark Streaming窗口计算示意图

每当窗口在源 `DStream` 上滑动时，落入该窗口内的源 `RDD` 被组合，并执行操作以产生窗口 `DStream` 的 `RDD`。在此特定情况下，该操作将应用于最后 3 个时间单位的数据，并以 2 个时间单位进行滑动。这表明任何窗口操作都需要指定两个参数。

- 窗口长度：窗口的持续时间（图 7-5 中的 3）。
- 滑动间隔：执行窗口操作的时间间隔（图 7-5 中的 2）。

这两个参数的值必须是源 `DStream` 的批间隔的倍数（图 7-5 中的 1）。

我们用一个例子来说明窗口操作。比方我们希望通过在过去 30 秒的数据中每 10 秒产生一次字数来扩展前面的例子。这是通过执行 `reduceByKeyAndWindow` 操作完成的。Spark Streaming 输出操作允许将 `DStream` 的数据推送到外部系统，如数据库或文件系统中。由于输出操作实际

上允许外部系统使用转换后的数据，因此它们会触发所有 DStream 转换的实际执行（类似于 RDD 的操作）。目前，Spark Streaming 支持的输出方法如表 7-3 所示。

表 7-3 Spark Streaming 支持的输出方法

方法	描述
print()	在运行流应用程序的驱动程序节点上的 DStream 中打印每批数据的前 10 个元素。这对开发和调试很有用
saveAsTextFiles(prefix, [suffix])	将此 DStream 中的内容保存为文本文件。每个批处理间隔的文件名都基于前缀和后缀“prefix-TIME_IN_MS [suffix]”生成
saveAsObjectFiles(prefix, [suffix])	将此 DStream 中的内容保存为序列化 Java 对象的 SequenceFiles。每个批处理间隔的文件名都基于前缀和后缀“prefix-TIME_IN_MS [suffix]”生成
saveAsHadoopFiles(prefix, [suffix])	将此 DStream 中的内容保存为 Hadoop 文件。每个批处理间隔的文件名都基于前缀和后缀“prefix-TIME_IN_MS [suffix]”生成
foreachRDD(func)	最通用的输出运算符，将函数 func 应用于从流中生成的每个 RDD 上。此功能应将每个 RDD 中的数据推送到外部系统，例如将 RDD 保存到文件，或者通过网络将其写入数据库。请注意，函数 func 在运行流应用程序的驱动程序进程中执行，并且通常会在其中执行 RDD 操作，这将强制计算流式 RDD

dstream.foreachRDD 是一个功能强大的原语，允许将数据发送到外部系统中。因此，了解如何正确有效地使用该原语非常重要。

通常将数据写入外部系统中需要创建连接对象（例如通过 TCP 连接到远程服务器），并使用它将数据发送到远程系统中。为此可能需要通过 Spark 驱动程序创建连接对象，然后把数据记录保存在 RDD 中。

7.2.6 优化运行时间

通过上文介绍我们知道，Spark Streaming 提供了方便的数据流、计算及结果输入方式，并且在容错等扩展方面提供了友好的处理方式。但是很有时候由于业务场景及数据量的不同，系统无法高效地处理外部的数据，因此还需要对 Spark 配置进行分场景优化。

1. 设置合理的批处理时间

时间分片对于 Spark Streaming 来说相当重要。因为 Spark 是按照时间分片提交作业的，如果时间分片设置得太小，就会造成频繁地提交任务，导致 Spark Streaming 吞吐量降低；如果设置得太大，则会造成任务的数据处理延时，以及对内存造成压力。理论上，Spark Streaming 的

batchDuration 可以达到毫秒级；实际上，在数据量大的情况下很难达到毫秒级。

2. 增加Job并行度

在设置计算的并行度时应尽可能利用集群的资源。如果是数据接入的性能不好，则可以考虑设置 inputStream 来提高接入的并行度

3. 使用Kryo序列化类

Spark 默认使用 Java 内置的序列化类，但是由于 java.io.Serializable 类的性能不佳，所以建议采用 Kryo 序列化类来替代。

4. 减少数据重复计算

对于反复使用的计算结果数据，为避免重复计算，建议采用数据缓存或者持久化方式，直接使用。

5. 设置合理的GC

Spark 是用 Scala 语言开发的，运行在 JVM 上就避免不了遇到因为内存不足导致的 GC 问题，不合理的 Java 的 GC 方式对 Spark 的性能有很大影响。对于 GC 方式可以研究 Java GC 方法，然后根据业务场景进行配置。

6. 设置合理的CPU数量

Spark 计算依赖 CPU 资源，为计算配置足够的 CPU 计算资源，让集群的资源得到更好的利用。可以通过 Spark Driver 和 Executor 来配置合理的 CPU 计算资源。

7.3 Flink计算框架

2008 年，Flink^[2]的前身已经是柏林理工大学的一个研究项目，2014 年被 Apache 孵化器接受，然后迅速成为 ASF（Apache Software Foundation）的顶级项目之一。截至本书写作时，Flink 的最新版本已经更新到 1.4.2。目前使用 Flink 的公司有阿里巴巴、Uber 等。

Flink 是一个针对流数据和批数据的分布式处理引擎，主要用 Java 代码实现。目前，Flink 主要还是依靠开源社区的贡献来发展的。对于 Flink，其处理的数据主要是流数据，批数据只是流数据的一个极限特例而已。Flink 的批处理方式采用的是流式计算原理，这一点跟 Spark 的设计思想正好相反（Spark Streaming 本质上是批处理，只是将计算分成了很小的单元，近似成流计算），这也是 Flink 的最大特点。Flink 支持本地快速迭代，以及一些环形的迭代任务。

7.3.1 基本概念

1. 数据集

数据集 (DataSet) 分为有界数据集和无界数据集。无界数据集的数据会源源不断地流入，有界数据集的数据是不可变的。许多传统上被认为是有界或“批”数据的真实数据集实际上是无界数据集。无界数据集包括但不限于：与移动或 Web 应用程序交互的最终用户、提供测量的物理传感器、金融市场、机器的日志数据。

2. 执行模型

实时处理是指当数据正在生成时连续执行的数据的处理过程。批处理是指在有限的时间内执行有限的数据的处理过程。不管采用哪种类型的执行模型来处理数据都是可以的，但却不一定是最优的。例如，批处理一直被应用于无界数据集的处理上，尽管它存在窗口、状态管理和次序错误等潜在问题。Flink 采用实时处理的执行模型，在数据处理精度和计算性能方面都有更大的优势。

3. Flink 程序模块

Flink 程序包含的主要模块有：Data Source、Transformations 和 Data Sink，如图 7-6 所示。其中，Data Source（数据源）就是要进入 Flink 处理的数据，如 HDFS、Kafka 中的数据等。Transformations 根据实际业务进行计算和转换。Data Sink 是 Flink 处理完的数据，即输出数据。

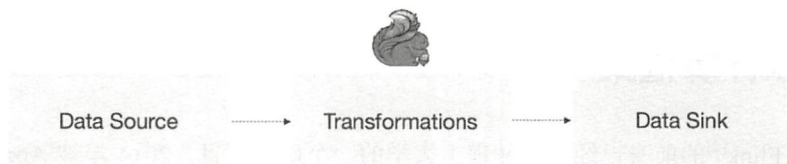


图7-6 Flink程序包含的主要模块

7.3.2 Flink 特点

Flink 是一个开源的分布式实时计算框架。Flink 是有状态的和容错的，可以在维护一次应用程序状态的同时无缝地从故障中恢复；它支持大规模计算能力，能够在数千个节点上并发运行；它具有很好的吞吐量和延迟特性。同时，Flink 提供了多种灵活的窗口函数。

1. 状态管理机制

Flink 检查点机制能保持 `exactly-once` 语义的计算。状态保持意味着应用能够保存已经处理的数据集结果和状态。Flink 的状态管理机制示意图如图 7-7 所示。

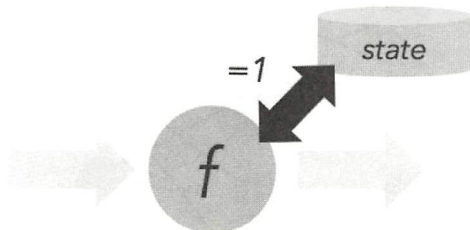


图7-7 Flink的状态管理机制示意图

2. 事件机制

Flink 支持流处理和窗口事件时间语义。事件时间可以很容易地通过事件到达的顺序和事件可能的到达延迟流中计算出准确的结果。Flink 的事件机制示意图如图 7-8 所示。

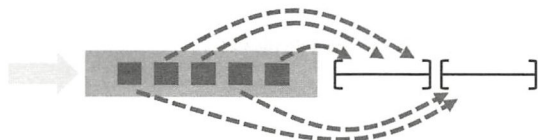


图7-8 Flink的事件机制示意图

3. 窗口机制

Flink 支持基于时间、数目以及会话的非常灵活的窗口机制 (window)。可以定制 window 的触发条件来支持更加复杂的流模式。Flink 的窗口机制示意图如图 7-9 所示。

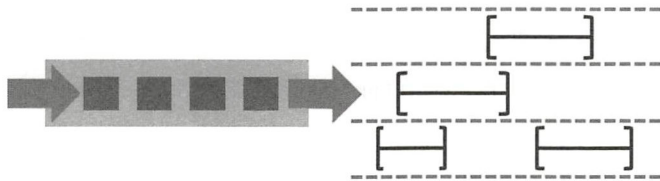


图7-9 Flink的窗口机制示意图

4. 容错机制

Flink 高效的容错机制允许系统在高吞吐量的情况下支持 `exactly-once` 语义的计算。Flink 可以准确、快速地做到从故障中以零数据丢失的效果进行恢复。Flink 的容错机制示意图如图 7-10 所示。

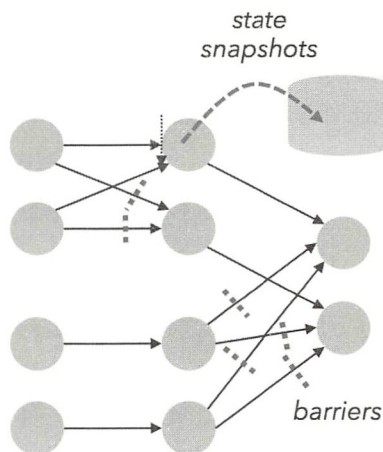


图7-10 Flink的容错机制示意图

5. 高吞吐量、低延迟

Flink 具有高吞吐量和低延迟（能快速处理大量数据）特性。图 7-11 展示了 Apache Flink 和 Apache Storm 完成分布式项目计数任务的性能对比。

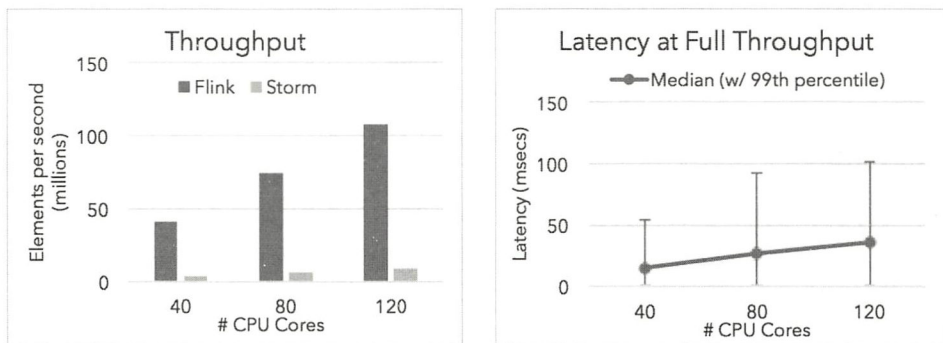


图7-11 Flink与Storm性能对比

6. 部署

可以通过 Yarn 和 Mesos 等资源管理软件来管理和部署 Flink。

7.3.3 运行原理

1. 链操作任务

分布式执行 Flink 的链操作任务，每个任务都由一个线程执行。将操作符链接到任务中是

一个有用的优化，其减少了线程间切换和缓冲的开销，并且在降低延迟的同时提高了总体吞吐量。可以配置链接行为，如图 7-12 所示。

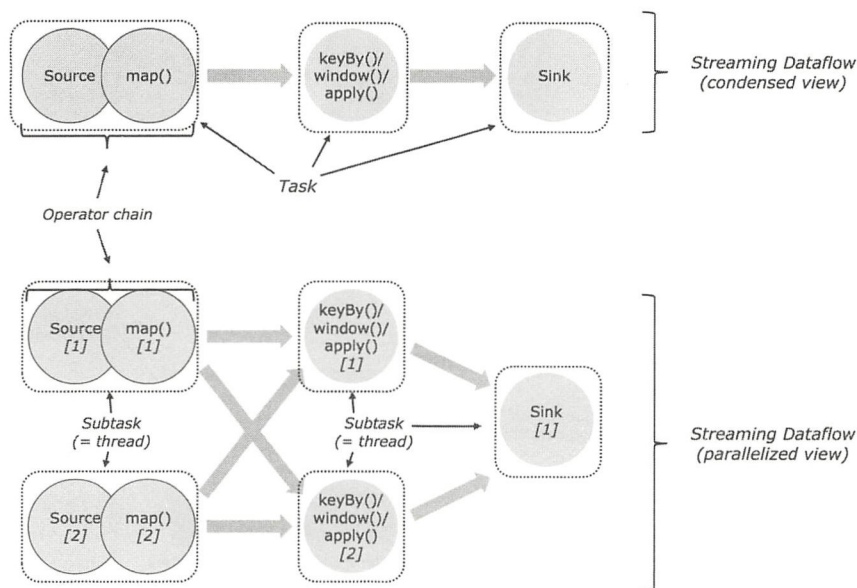


图7-12 Flink的链操作示意图

2. 任务提交

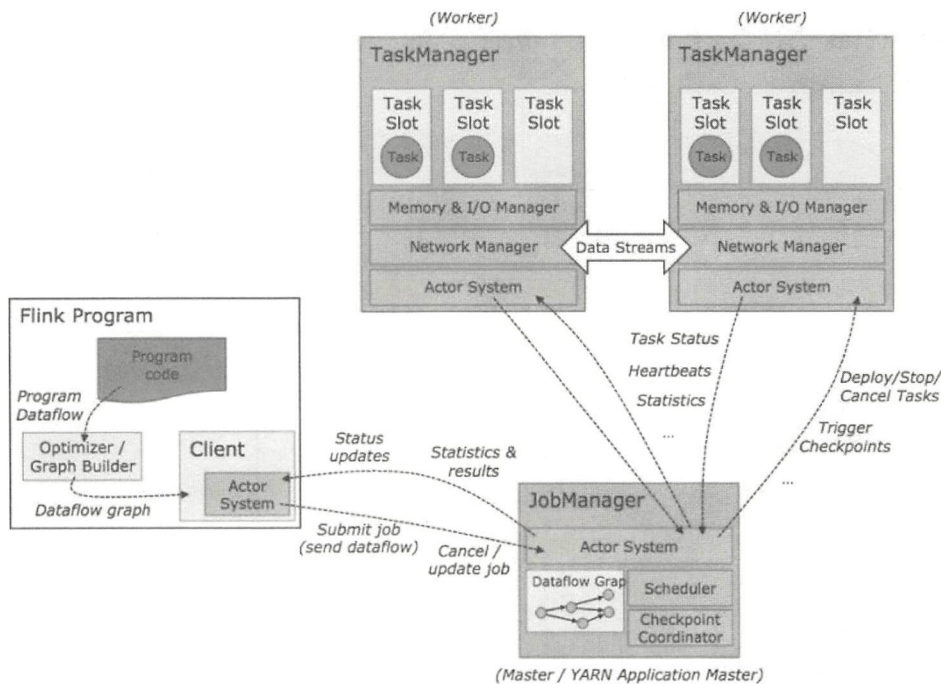
Job Tracker: 协调分布式执行——安排任务、协调检查点、协调故障恢复等。为了具有高可用性，设置了多个 `JobManager`，其中一个领导者，其他的作为备用。

Task Tracker: 执行任务（更具体地说，是一个数据流任务）、和缓冲区交换数据流。

Client: 客户端用来进行任务调度前期的准备（数据、环境变量等），然后提交计算任务到 `JobManager`。任务提交之后，客户端可以断开连接，也可以继续保持连接以接收进度报告。

3. 运行

当 Flink 集群启动后，首先会启动一个 `JobManager` 和一个或多个 `TaskManager`。由客户端提交任务给 `JobManager`，`JobManager` 再调度任务到各个 `TaskManager` 来执行，然后 `TaskManager` 将心跳和统计信息汇报给 `JobManager`。`TaskManager` 之间以流的形式进行数据传输，如图 7-13 所示。

图7-13 Flink整体流程图（图片来源于Flink官网^[2]）

4. 任务槽和资源

每个 Worker (TaskManager) 都是一个 JVM 进程，并且可以在单独的线程中执行一个或多个子任务。为了控制 Worker 可以接收多少个任务，Worker 有所谓的任务槽（至少一个）。

每个任务槽都代表 TaskManager 的一个固定资源子集。例如，具有三个插槽的 TaskManager 将为每个插槽分配 1/3 隔离的内存资源，这意味着子任务不会与其他作业中的子任务来竞争内存。请注意，目前插槽仅分离托管的任务内存，不会进行 CPU 的隔离。。

通过调整任务槽的数量，用户可以定义子任务如何彼此隔离。每个 TaskManager 都拥有一个插槽，这意味着每个任务组都可以在单独的 JVM 中运行（例如，可以在单独的容器中启动）；而拥有多个插槽，则意味着更多的子任务共享相同的 JVM。同一个 JVM 中的任务共享 TCP 连接（通过多路复用）和心跳消息，它们也可能共享数据集和数据结构，从而减少每个任务的开销。Flink 的任务槽示意图如图 7-14 所示。

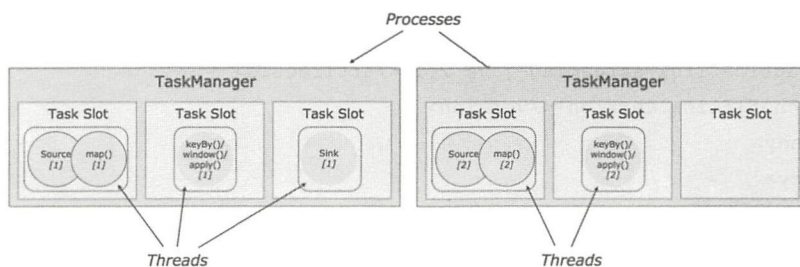


图7-14 Flink的任务槽示意图

7.3.4 Java API 的使用

本节将通过 Flink 官网上的例子，从建立 Flink 项目开始，在 Flink 群集上运行流式分析程序。

维基百科提供了一个 IRC 频道，其中所有对 Wiki 的编辑都被记录下来。我们将在 Flink 中读取此频道，并计算每个用户在给定窗口时间内编辑的字节数。使用 Flink 很容易实现，并为构建更复杂的分析程序提供了良好的基础。

1. 创建Maven工程

```
$ mvn archetype:generate \
  -DarchetypeGroupId=org.apache.flink \
  -DarchetypeArtifactId=flink-quickstart-java \
  -DarchetypeVersion=1.4.2 \
  -DgroupId=wiki-edits \
  -DartifactId=wiki-edits \
  -Dversion=0.1 \
  -Dpackage=wikiedits \
  -DinteractiveMode=false
```

2. 增加Maven依赖

```
<dependencies>
  <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-java</artifactId>
    <version>${flink.version}</version>
  </dependency>
```

```

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.11</artifactId>
  <version>${flink.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-clients_2.11</artifactId>
  <version>${flink.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-wikiedits_2.11</artifactId>
  <version>${flink.version}</version>
</dependency>
</dependencies>

```

3. 开发程序

```

package wikiedits;

import org.apache.flink.api.common.functions.FoldFunction;
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.connectors.wikiedits.WikipediaEditEvent;
import org.apache.flink.streaming.connectors.wikiedits.WikipediaEditsSource;

public class WikipediaAnalysis {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment see = StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<WikipediaEditEvent> edits = see.addSource(new WikipediaEditsSource());

        KeyedStream<WikipediaEditEvent, String> keyedEdits = edits

```

```

        .keyBy(new KeySelector<WikipediaEditEvent, String>() {
            @Override
            public String getKey(WikipediaEditEvent event) {
                return event.getUser();
            }
        });

    DataStream<Tuple2<String, Long>> result = keyedEdits
        .timeWindow(Time.seconds(5))
        .fold(new Tuple2<>("", 0L), new FoldFunction<WikipediaEditEvent, Tuple2<String,
Long>>() {
            @Override
            public Tuple2<String, Long> fold(Tuple2<String, Long> acc, WikipediaEditEvent event) {
                acc.f0 = event.getUser();
                acc.f1 += event.getBytesDiff();
                return acc;
            }
        });

    result.print();

    see.execute();
}
}

```

4. 本地运行

```

$ mvn clean package
$ mvn exec:java -Dexec.mainClass=wikiedits.WikipediaAnalysis

```

5. 结果展示

```

1> (Fenix down,114)
6> (AnomieBOT,155)
8> (BD2412bot,-3690)
7> (IgnorantArmies,49)
3> (Ckh3111,69)
5> (Slade360,0)
7> (Narutolovehinata5,2195)
6> (Vuyisa2001,79)
4> (Ms Sarah Welch,269)
4> (KasparBot,-245)

```

6. 程序解释

Flink 程序首先创建了一个 `StreamExecutionEnvironment`（如果你正在编写批处理作业，则为 `ExecutionEnvironment`），用来设置执行参数并创建来自外部系统的读取源。我们将其添加到主要方法中：

```
StreamExecutionEnvironment see = StreamExecutionEnvironment.getExecutionEnvironment();
//创建 WikipediaEditEvent 类型的 DataStream
DataStream<WikipediaEditEvent> edits = see.addSource(new WikipediaEditsSource());
//因为要计算每个用户在给定窗口时间内编辑的字节数，所以需要用户对用户分组，将同一个用户的数据分组到
//同一个流中进行计算，这里采用 KeyBy 的转换通过用户名进行分组
KeyedStream<WikipediaEditEvent, String> keyedEdits = edits
    .keyBy(new KeySelector<WikipediaEditEvent, String>() {
        @Override
        public String getKey(WikipediaEditEvent event) {
            return event.getUser();
        }
    });
//我们已经将用户按照用户名分组了，现在按照 5 秒的时间窗口进行分用户统计，窗口采用折叠转换模式
//对每一个用户在这个时间窗口内的编辑字节数据进行计算
DataStream<Tuple2<String, Long>> result = keyedEdits
    .timeWindow(Time.seconds(5))
    .fold(new Tuple2<>("", 0L), new FoldFunction<WikipediaEditEvent, Tuple2<String, Long>>() {
        @Override
        public Tuple2<String, Long> fold(Tuple2<String, Long> acc, WikipediaEditEvent event) {
            acc.f0 = event.getUser();
            acc.f1 += event.getByteDiff();
            return acc;
        }
    });
```

接下来要做的就是将流打印到控制台并开始执行。

```
result.print();
see.execute();
```

7.4 案例分析

广告在任何互联网公司都是核心业务，直接关系到公司的收入。微博广告收入占微博收入

的 80%，为此广告投放系统的健康度及投放效果的重要性就相当突出了。微博广告分为保量广告、竞价广告、效果广告等包含超粉、品牌广告、涨粉、非粉、粉丝头条等广告产品。微博广告的请求量在百亿级别以上，并且涉及的产品线较多。我们需要对微博产品各个产品线的请求、曝光、互动、结算等数据进行实时监控。

7.4.1 背景介绍

1. 监控数据如何采集及技术选型

数据采集是数据计算的第一个环节，准确地采集数据是计算的基础。微博广告数据的采集主要采用 Filebeat 来完成。Filebeat 是一个日志文件托运工具，在服务器上安装客户端后，Filebeat 会监控日志目录或者指定的日志文件，追踪读取这些文件（追踪文件的变化，不停地读），并且将这些信息转发到消息队列、Elasticsearch 或者 Logstash 中存放。

2. 数据如何传输及技术选型

数据传输采用 Kafka 消息中间件来完成。Kafka 是 LinkedIn 于 2010 年 12 月开发并开源的一个分布式流平台，现在是 Apache 的顶级项目，是一个高性能的、跨语言的、分布式的发布/订阅消息队列系统，消费者通过拉取方式消费消息。Kafka 消息中间件的特性：快速持久化，可以在 $O(1)$ 的系统开销下进行消息持久化；高吞吐量，在一台普通的服务器上即可以达到 10W/s 的吞吐速率；完全的分布式系统，Broker、Producer、Consumer 都原生自动支持分布式，自动实现复杂均衡。因为在设计之初 Kafka 是作为日志流平台和运营消息管道平台来使用的，所以实现了消息顺序和海量存储。使用 Kafka 消息队列极大地提高了计算的横向扩展能力。

3. 数据如何计算及技术选型

计算引擎通过消息队列的数据进行计算。微博广告有自研的计算引擎 OLS、阿里巴巴的 JStorm、Spark Streaming、Flink 等实时引擎。由于 Flink 具有低延迟、高吞吐量、高度灵活的时间窗口的特性，所以最终采用 Flink 计算引擎。

4. 数据如何存储及技术选型

数据存储决定了响应的速度，存储可选择 HBase、Druid、ClickHouse。HBase 的写入吞吐量高，但是查询性能不能胜任；相对于 HBase，Druid 的查询比较灵活，但是在单个维度枚举值范围过大的情况下，查询性能也是不可忍受的；ClickHouse 支持实时写入，并且拥有超强的查询性能。所以最终选择采用 ClickHouse 存储。

7.4.2 架构设计

综合考虑数据采集、传输、计算，因此整体架构采用分布式系统设计，支持横向扩展、容错。消息队列和存储都有备份机制来防止数据丢失，计算引擎 Flink 的出色的计算能力可以更好地服务于计算，查询引擎 ClickHouse 能够在计算结果上进行多维度的业务分析。

7.4.3 效果分析

在可靠而又稳定的监控报警平台支持下，2017 年广告的监控报警指标快速增长，监控指标增幅超过 100%。监控业务包括超粉、品速、粉丝经济、结算等；监控维度包括平台、场景、竞价类型、地域、性别、年龄、创意类型等；监控指标包括曝光、请求、互动、CPM、ECPM、CPE 等。

7.5 本章小结

本章主要对实时计算进行了简要说明，对实时计算的实时性、准确性、响应速度进行了阐述。本章还介绍了 Spark Streaming 和 Flink 分布式实时计算引擎的背景、基本概念、运行原理及简单使用。通过对微博广告监控实时应用的架构进行分析，可以更好地帮助读者理解实时计算。

7.6 参考文献

[1] Spark 官网：<http://spark.apache.org/docs/latest/streaming-programming-guide.html>

[2] Flink 官网：<http://flink.apache.org/>

第 8 章

时序数据分析框架

8.1 时序数据库简介

8.1.1 什么是时序数据库

在介绍时序数据库之前，我们先来看看什么是时序数据。时序数据就是基于时间排序的数据，再通过时间坐标将这些数据连接起来，形成一个折线图，直观地展示一个指标在过去一段时间内的走势和规律，帮助定位数据异常点。

时序数据库就是用来存储这些时序数据的数据库。与传统数据库相比，时序数据库需要能够长时间保存数据，且需要实时展示，这就要求时序数据库能做到持久化存储，以及数据读写的高性能。此外，对于一些复杂的场景，比如广告业务的多维度多指标，时序数据库还需要做到多维查询、指标聚合等。图 8-1 展示了某服务最近 6 小时的 QPS 走势。

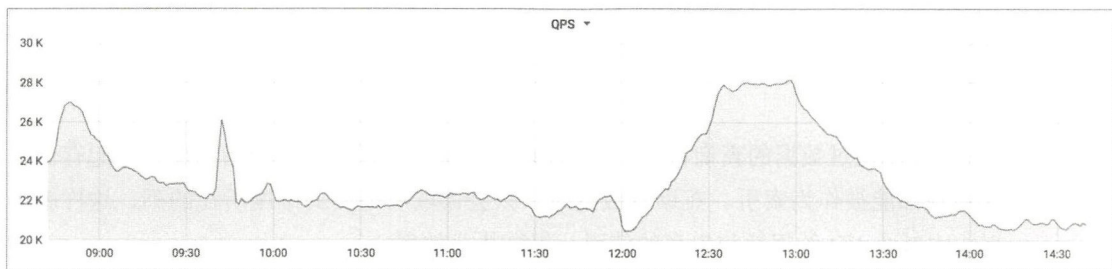


图8-1 QPS走势图

8.1.2 时序数据库的特点

1. 数据写入

- 数据实时写入。
- 高并发写入。
- 无须更新或删除操作（除了修复数据）。
- 连续性：时序数据会按照指定的时间粒度持续写入。

2. 数据读取

- 写多读少：时序数据的写入是持续的，但是一般并不会持续地读取数据，只有在需要的时候才会查询最近一段时间内的指定维度指标。
- 多时间粒度读取：一般来说，我们会对最近 7 天的数据以一个比较细的时间粒度来存储一个相对精确的值，而对于 7 天或者一个月以前的数据，通常会把它们聚合成一个比较粗的时间粒度存储，比如按照小时或者天来存储，以便节省磁盘空间，提高读取历史数据的效率。
- 指定维度读取：在广告业务中，时序数据存储的维度可能有成百上千个，那么在读取数据时不会把所有的维度都读取出来，因为这样做不仅没必要，而且对系统的 I/O 也是一个极大的考验。所以，只会选择读取所需要的维度和指标。
- 实时聚合：通常实时的时序数据存储的是不同维度下的一个比较细的时间粒度数据，查询时需要在不同维度下对一个或多个指标进行各种聚合的操作，如 `sum`、`max`、`avg` 等。

3. 数据存储

- 按列存储：通过数据的查询特征，可以发现时序数据更适合将一个指标放在一起存储，任何列都能被作为索引。在读取数据时，只会读取所需要的维度所在的列，这样就可以大大减小 I/O 的损耗和内存的使用，提高执行效率。
- 以不同的时间粒度存储：数据的读取特征决定了可以将历史数据聚合成一个比较粗的粒度存储，将最近的数据以一个比较细的粒度存储，这样可以大大减少磁盘的使用空间。

- 冷热存储：通常我们只会查询最近一天或者 7 天的数据，而半年或一年以前的数据使用率很低，因此可以把历史的冷数据和最近的热数据分开存储，以提高读取数据的效率，减少磁盘的使用空间。

4. 时序数据库的特性

上面介绍的时序数据库的这些特点，决定了时序数据库具有以下特性。

- 高并发、高吞吐量，实时写入和读取数据。
- 高可用性、高可靠性，分布式架构、数据分片。
- 支持海量数据存储，一般时序数据都是 TB 或 PB 级别的体量。
- 支持数据聚合分析，满足实时的多维聚合分析。

5. 时序数据库的组成（不同的数据库定义可能不一样）

- Timestamp：这是时序数据库的关键所在，因为是以时间排序的数据，所以需要记录所有数据的时间。
- Metric：需要存储的指标数据有很多，比如在广告业务中，分析用到的指标更是成千上万，那么在时序数据库中就需要通过 Metric 字段来标识每个指标数据，如 QPS、Status 等
- Dimension：数据的属性，比如数据类型、地域、年龄、性别等。而一般指标的结果都是基于单维度或多维度分析得到的。比如在广告业务中，一个指标有时需要上百个维度经过不同的组合得到不同的结果。

6. 时序数据库的模型

表 8-1 显示了在同一个时间、不同维度下 QPS 的变化情况。其中地域和性别就是 Dimension，北京、上海等就是地域维度的类别，QPS 就是 Metric，QPS 列的数据就是 QPS 指标每秒钟在地域和性别组合维度下的值。

表 8-1 时序数据库模型

Timestamp	地域	性别	QPS
1522166400	北京	男	15000
1522166400	北京	女	13000
1522166400	上海	男	18000
1522166400	上海	女	11000
1522166400	广州	男	12000

续表

timestamp	地域	性别	QPS
1522166400	广州	女	11000
1522166400	成都	男	8000
1522166400	成都	女	9000

8.1.3 时序数据库的对比

目前，市面上的时序数据库种类繁多，比较老牌的有 Graphite、RRD Tool 等，后起之秀有 InfluxDB、OpenTSDB、Prometheus 等，还有一些虽然不能定位为时序数据库，但是可以轻松实现时序数据存储的功能，支持海量数据做实时聚合运算，且功能强大，比如 Druid、Elasticsearch、ClickHouse 等。表 8-2 对比了目前主流时序数据库的优缺点。

表 8-2 主流时序数据库的优缺点

时序数据库	优点	缺点
Graphite	<ul style="list-style-type: none">提供丰富的函数支持支持自动 Downsample对 Grafana 的支持最好维护简单	<ul style="list-style-type: none">Whisper 存储引擎 IOPS 高Carbon 组件 CPU 使用率高聚合分析功能弱
InfluxDB	<ul style="list-style-type: none">Metric+Tags部署简单、无依赖实时数据 Downsample高效存储	<ul style="list-style-type: none">开源版本没有集群功能存在前后版本兼容问题存储引擎在变化聚合分析功能弱
OpenTSDB	<ul style="list-style-type: none">Metric+Tags集群方案成熟（HBase）写高效（LSM-Tree）	<ul style="list-style-type: none">查询函数有限依赖 HBase运维复杂聚合分析功能弱
Prometheus	<ul style="list-style-type: none">Metric+Tags适用于容器监控具有丰富的查询语言维护简单集成监控和报警功能	<ul style="list-style-type: none">没有集群解决方案聚合分析功能弱

续表

时序数据库	优点	缺点
Druid	<ul style="list-style-type: none"> 支持嵌套数据的列式存储 具有强大的多维聚合分析能力 实时高性能数据摄取 具有分布式容错架构 支持类 SQL 查询 	<ul style="list-style-type: none"> 一般不能查询原始数据 不适合维度基数特别高的场景 时间窗口限制了数据完整性 运维较复杂
Elasticsearch	<ul style="list-style-type: none"> 具有强大的多维聚合分析能力 支持全文检索 支持查询原始数据 灵活性高 社区活跃 扩展丰富 	<ul style="list-style-type: none"> 不支持分析字段的列式存储 对硬件资源要求高 集群维护较复杂
ClickHouse	<ul style="list-style-type: none"> 具有强大的多维聚合分析能力 实时高性能数据读写 支持类 SQL 查询 提供丰富的函数支持 具有分布式容错架构 支持原始数据查询 适用于基数大的维度存储分析 	<ul style="list-style-type: none"> 比较年轻, 扩展不够丰富, 社区还不够活跃 不支持数据更新和删除 集群功能较弱

8.2 时序数据库Graphite

8.2.1 Graphite 简介

Graphite^[1]项目创建于 2006 年, 算是比较老的时序数据库了。它是使用 Python 语言编写的, 社区活跃, 文档详细。

Graphite 主要由三个模块组成。

(1) Whisper: Graphite 中的时间序列数据库。

(2) Carbon: 接收数据写入请求、数据缓存、数据聚合、数据分区与复制。

○ carbon-relay: 复制和分片。

- carbon-aggregator: 数据聚合。
- carbon-cache: 数据缓存, 接收指标存储在缓存中, 定期写入。

(3) Graphite-Web: 读取、展示数据的 Web 端。

Graphite 架构示意图如图 8-2 所示。

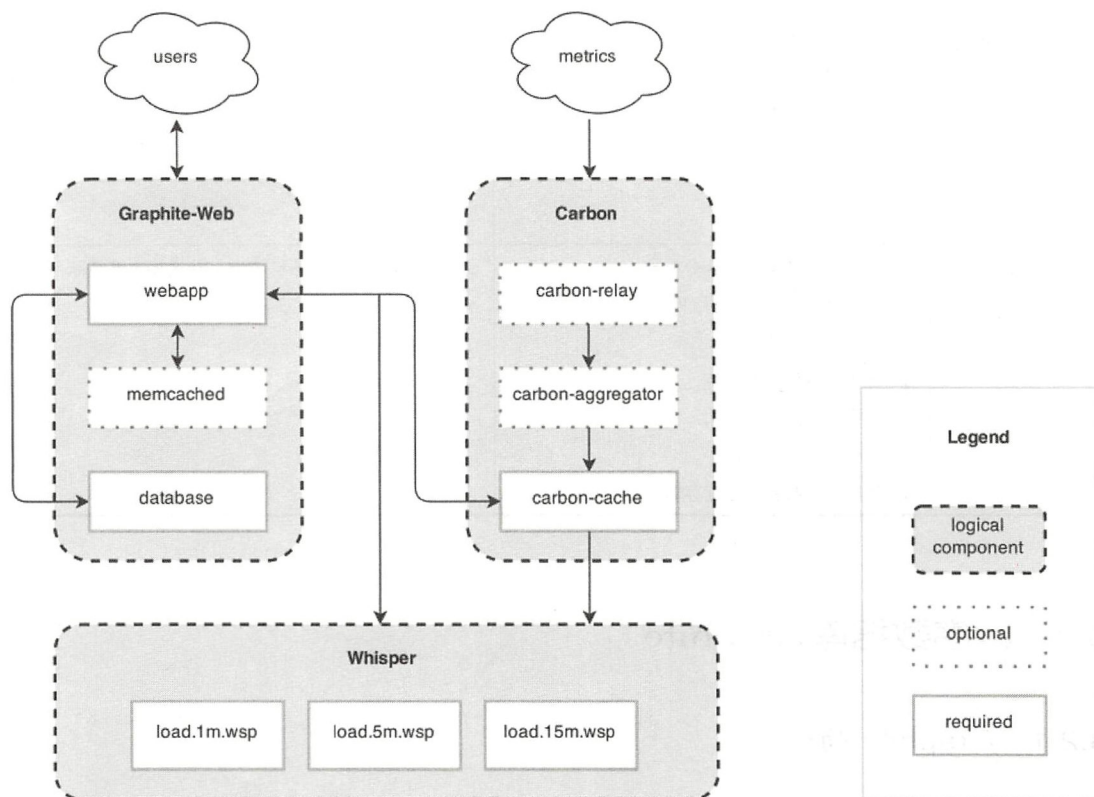


图8-2 Graphite架构示意图

1. carbon-relay

数据转发: relay 可以用来转发数据到不同的 carbon-aggregator 或 carbon-cache 上, 以实现数据分片的目的

数据过滤: 可以在 relay-rules.conf 中通过正则表达式指定数据指标的目的 carbon-cache 地址。


```
[nginx]
pattern=^nginx.status.*
servers=192.168.1.1:2004,192.168.1.2:2004
```

2. carbon-aggregator

carbon-aggregator 是 Graphite 的聚合进程, 数据在通过 relay 转发分片后, 通过 carbon-aggregator 聚合, 再发送给 carbon-cache 写入 Whisper。聚合后的数据的时间粒度比较粗, 但是可以降低 I/O 的损耗和减小所占用的磁盘空间。配置过滤的格式如下:

```
output_template (frequency) = method input_pattern
```

carbon-aggregator 捕获所有匹配 input_pattern 的指标, 并在指定的 frequency 时间内通过指定的聚合方法 method, 对捕获的数据进行聚合。聚合后的指标会填充到 output_template 指定的地方。

目前, carbon-aggregator 支持的聚合方法包括: sum、avg、min max、p50、p75、p80、p90、p95、p99、p999 和 count。

3. carbon-cache

carbon-cache 是 Graphite 的缓存进程。carbon-cache 接收多种协议。carbon-cache 将接收到的指标数据放到 RAM 中, 并定时将它们刷到 Whisper 中持久存储。carbon-cache 还支持把热数据缓存在内存中来查询, 这极大地提高了热数据的查询效率。

carbon-cache 通过配置文件 storage-schemas.conf 来描述存储指标的保留频率。它将指标的路径与 storage-schemas.conf 中定义的正则表达式相匹配, 从而通知 Whisper 每个指标的存储频率和历史周期。

需要注意的是, 如果改变了 storage-schemas.conf 文件中定义的规则, 那么对于已经在 Whisper 中生成的.wsp 文件是无效的, 只能通过 Whisper 提供的 whisper-resize.py 脚本来改变已经生成的.wsp 文件。

storage-schemas.conf 中的每个规则都通过三个元素来指定。

- 规则名称。
- 匹配指标的正则表达式。
- 保留区: 对于指标保留的时间和频率, 可以指定多个组合。

比如我们定义了这样一个 Nginx 规则：

```
[nginx]
pattern = ^nginx.status*
retentions = 1s:30m,1m:180d,1d:720d
```

第一行：nginx 为规则名称。

第二行：pattern 为匹配指标的正则表达式，它将匹配所有以 nginx.status 开头的指标。

第三行：定义了指标的三个保留区，每个保留区都是以 frequency:history 的样式呈现的，用逗号分隔每个保留区。这里表示以 nginx.status 开头的指标会有三个保留区，其中第一个保留区表示按秒粒度存储指标数据，并且将会保留最近 30 分钟的数据；第二个保留区表示按分钟粒度存储指标数据，并且将会保留最近 180 天的数据；第三个保留区表示按天粒度存储指标数据，并且将会保留最近 720 天的数据。保留区中的时间单位可以是如下几种情况：

```
s — second
m — minute
h — hour
d — day
w — week
y — year
```

4. Whisper

Whisper 是 Graphite 中默认的时序数据库，与 RDD-Tool 的存储方式类似。

(1) Whisper 的数据存储

Whisper 在一个指标被创建时就会固定指标文件的格式和大小。一个 Whisper 指标文件分为两部分内容：元数据和指标数据。

元数据部分描述了该指标文件有几个保留区，以及每个保留区的偏移值、时间粒度、数据点数、尺寸大小等信息，如下所示。

```
Meta data:
  aggregation method: sum
  max retention: 62208000
  xFilesFactor: 0

Archive 0 info:
  offset: 52
```

```
seconds per point: 1
points: 1800
retention: 1800
size: 21600
```

Archive 1 info:

```
offset: 21652
seconds per point: 60
points: 259200
retention: 15552000
size: 3110400
```

Archive 2 info:

```
offset: 3132052
seconds per point: 86400
points: 720
retention: 62208000
size: 8640
```

这里定义了三个保留区，每个保留区的时间粒度和存储周期如下：

1. 按秒存储，周期为 30 分钟。
2. 按分钟存储，周期为 180 天。
3. 按天存储，周期为 720 天。

指标数据部分按照保留区存储每个时间粒度下的指标值。每个保留区都分为两列：时间戳和指标值。如下展示了第一个保留区中 10 秒的指标数据。

Archive 0 data:

```
0: 1518148666,      2
1: 1518166667,      1
2: 1518164868,      1
3: 1518166669,      1
4: 1518163070,      1
5: 1518157671,      1
6: 1518164872,      2
7: 1518164873,      1
8: 1518163074,      1
9: 1518166675,      1
```

每个保留区数据点都会按照指定的时间周期被重复使用。比如，上例中第一个保留区是按秒存储的，周期为 30 分钟，那么 Whisper 会事先在这个指标文件中创建 1800 个数据点，并占用相应的存储空间，当指标数据写满 1800 个数据点后，又会从头开始覆盖写入新的数据。

Carbon 默认分为两个保留区：60s 存储 180d 和 1d 存储 720d。

（2）Whisper 的数据聚合

上面介绍了 Whisper 的数据是如何存储的，那么按秒存储的数据如何变成按分钟或者按天存储的呢？

按照上面的例子，还是分成三个保留区 1、2、3。Whisper 用第二个保留区的时间粒度 60s 整除第一个保留区的时间粒度 1s，得到 60 个数据点，那么 Whisper 会聚合第一个保留区的前 60 个数据点的数据，写入第二个保留区的第一个数据点位上。当第二个保留区写入 1440 个数据时，就会触发将这 1440 个数据值聚合写到第三个保留区的数据点位上。

Whisper 还通过有效数据点占比来提高数据的准确性。在 carbon-aggregator 的配置文件 storage-aggregation.conf 中，有一个参数叫 xFilesFactor，可以通过这个参数给指定的指标数据设置一个有效数据点比例，只有当有效数据数量达到这个比例时，才对数据进行聚合。比如，上面说第一个保留区向第二个保留区聚合时需要 10 个数据点，当设置 xFilesFactor 为 0.5 时，那么在这 10 个数据点中至少要有 5 个数据点有有效数据才能聚合，如果只有 4 个数据点有有效数据，其余 6 个数据点都是空值，那么在第二个保留区聚合的数据点位上只会写入空值。

Whisper 支持的聚合函数有 min、max、sum、average、last。

Whisper 的文件格式为.wsp，我们需要通过 Whisper 自身携带的 whisper-dump.py 脚本来查看每个数据文件中的内容。

（3）Whisper 的劣势

Whisper 在数据精度控制和时间粒度方面拥有良好的表现，但是它的缺点也很明显。

- 性能差。
- 在大量请求下 IOPS 占用高。
- 指标文件一旦生成格式就固定下来了，不能更改。
- 不支持灵活的多维组合分析。

虽然可以通过 SSD、数据分片来优化性能，但是指标的灵活性和多维的组合分析依然是它的瓶颈。

5. Graphite-Web

Graphite-Web 是 Graphite 提供的前端组件，它可以可视化我们存储的指标数据，以直观的图形页面呈现数据的走势。我们还可以通过 Graphite-Web 提供的 render API 接口来获取原始数据。

但是 Graphite-Web 的配置复杂，功能不完善，可视化的美观程度远远不及其他开源的可视化工具。相比较而言，开源的可视化工具 Grafana 具有优美的外观、强大的图形展示和编辑功能，以及灵活的函数调用，让我们不得不放弃 Graphite-Web 而投入 Grafana 的怀抱。

8.2.2 Graphite 在广告监控系统中的应用

在写入量比较大的环境中，默认的 Python 版 Carbon 的性能存在瓶颈，所以我们替换成开源的 Go 版 Carbon 来提高数据读写的性能。并且使用内存模式来存储最近一天的指标数据，而将一天以上的数据存储在 SSD 上。通过 Graphite-Web 按读取时间来路由所读取的数据目录，最近一天的数据从内存中读取，一天以上的数据从 SSD 上读取。下面介绍 Go 版 Carbon 的安装方式。

1. 挂载内存

首先将内存挂载到文件系统中。当然，需要预留一定的内存给系统的其他应用使用。比如线上服务器拥有 128GB 的内存，可以使用 80GB 的内存作为文件系统来缓存热数据。

```
# 挂载内存到文件系统中
mount -t tmpfs -o noatime,size=80G,nr_inodes=0 tmpfs /opt/graphite
# 软链接到根目录
ln -s /opt/graphite /
```

2. 安装Carbon

```
git clone https://github.com/lomik/go-carbon.git
cd go-carbon/
make submodules
make
install -m 0755 go-carbon /usr/local/bin/go-carbon
```

3. 配置内存模式Carbon

(1) 配置主配置文件

```
[common]
# 配置运行 Carbon 服务的用户
user = "graphite"
# 定义 Carbon 内部性能指标的监控数据存储路径
graph-prefix = "carbon.agents.{host}"
# 定义 Carbon 内部性能指标的位置
metric-endpoint = "local"
# 定义 Carbon 内部性能指标的存储时间粒度
metric-interval = "10s"
# 定义使用的 CPU 核数
max-cpu = 8

[whisper]
# 定义 Whisper 在内存模式下的数据目录
data-dir = "/data0/graphite/storage/whisper/"
# 定义内存模式下的 carbon-schemas 配置文件的位置
schemas-file = "/etc/carbon-mem-schemas"
# 定义 carbon-aggregation 配置文件的位置
aggregation-file = "/etc/carbon-aggregation"
# 定义 Whisper 的工作线程数
workers = 8

[tcp]
# 配置 Carbon 的端口
listen = ":3003"
# 开启 TCP 模式
enabled = true
# 接收端和缓存之间的队列大小
buffer-size = 0
```

(2) 配置 Carbon Schemas

```
[nginx]
pattern = ^nginx.status.*
# 这里配置内存模式下的数据存储规则，这是一个热数据存储，因此要求时间粒度比较细，但是不要求存储很长时间
retentions = 1s:30m
```



(3) 配置 Carbon Aggregation

```
[nginx]
pattern = ^nginx.status.*
xFilesFactor = 0.8
aggregationMethod = sum
```

4. 配置硬盘模式Carbon

硬盘的配置文件和内存的配置文件相比，有以下两个地方不一样。

- Whisper 的数据目录：在硬盘模式下，需要指定持久化在 SSD 中的 Whisper 数据目录路径。
- schemas-file 配置文件：定义硬盘模式下的 schemas-file 配置文件的位置。配置如下：

```
[nginx]
pattern = ^nginx.status.*
# 配置硬盘模式下的数据存储规则，这是一个持久化的数据存储，因此时间粒度相比内存而言要求更粗，时间更长
retentions = 10s:1d:60s:180d:86400s:720d
```

5. 启动Carbon

```
# 启动内存模式 Carbon
go-carbon --config /etc/carbon_mem.conf --daemon
# 启动硬盘模式 Carbon
go-carbon --config /etc/carbon_ssd.conf --daemon
```

8.3 多维分析利器Druid

8.3.1 什么是 Druid

Druid^[2]是一个用于大数据实时查询与分析的分布式列式数据存储系统。为了应对海量数据的实时查询和多维分析，Druid 应运而生。

Druid 诞生于 MetaMarkets 公司，而互联网广告分析正是 MetaMarkets 最重要的业务之一，基于同样的业务需求背景，微博广告也开始尝试将 Druid 作为监控平台后端数据引擎的技术方案之一。当然，Graphite 时序数据库在多维分析上的劣势、在集群方案上的缺陷等原因，也让我们开始寻找更适合的时序数据库来替换 Graphite。



Druid 的特性如下：

- 支持部分嵌套数据结构的列式存储。
- 进行剪枝的分布式层级查询。
- 通过索引快速过滤。
- 数据的实时摄入和查询。
- 容错分布式架构确保数据不丢失。
- 水平扩展。

8.3.2 Druid 架构

1. Druid节点组成

Druid 是一个分布式系统，由多个角色的节点组合而成，每个节点都只关心自己的工作。Druid 架构示意图如图 8-3 所示。

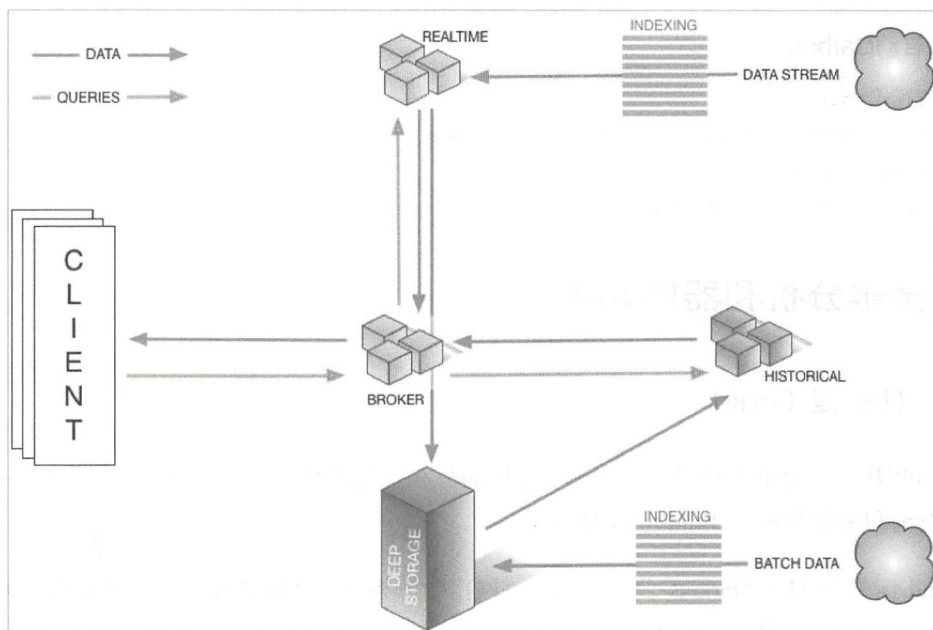


图8-3 Druid架构示意图

- **实时节点**：实时节点提供实时的索引服务，通过这些节点索引的数据可以立即用于查询。实时节点将在一定时间内收集到的数据生成一个 **Segment** 数据文件，并将这个



Segment 数据文件发送到历史节点中。通过 ZooKeeper 监控传输和元数据库来存储被发送的 Segment 数据文件的元数据。发送完成后，实时节点就会丢弃被发送的 Segment 数据文件。

- **历史节点：**存储和查询历史数据。历史节点不与其他节点直接通信，而是通过与 ZooKeeper 保持一个长连接，实时观察指定路径下的新 Segment 数据文件信息。当历史节点注意到 ZooKeeper 指定队列路径下有一个条目时，首先它会检查本地缓存中是否存在该 Segment 数据文件的元信息，如果本地缓存中不存在新 Segment 数据文件的元信息，那么历史节点将从 ZooKeeper 中读取新 Segment 数据文件的元信息到本地，并根据下载到本地的 Segment 数据文件的元信息去 Deep Storage 中拉取 Segment 数据文件。最后，历史节点会通过 ZooKeeper 向集群宣布，它将提供该 Segment 数据文件的查询服务。
- **协调节点：**主要负责 Segment 数据文件的加载、删除，管理 Segment 数据文件副本和平衡各历史节点的 Segment 数据文件。协调节点根据配置文件中指定的参数定期运行，每次运行，它都会监测集群状态，并采取相关的动作。协调节点通过 ZooKeeper 来获取集群信息，同时还会保持与存储可用段表和规则表的数据库的连接。协调节点不会直接与历史节点通信，它会根据规则、容量等信息指定历史节点，并在 ZooKeeper 中该历史节点的队列路径下生成关于 Segment 数据文件的临时信息。历史节点看到这些临时信息，就会去拉取 Segment 数据文件。
- **代理节点：**接收客户端的查询请求，并路由请求到实时节点或历史节点。代理节点通过 ZooKeeper 知道每个 Segment 数据文件存在于哪个节点上。代理节点还用于聚合每个节点返回的结果，再将聚合后的结果集返回给客户端。代理节点还可以通过配置缓存来存储历史节点查询的结果，提高了相同查询的效率。
- **索引服务节点：**索引服务是一个高可用的分布式服务，运行与索引相关的任务。索引服务是一个主/从架构。索引服务由三个组件组成，即运行一个任务的 peon 组件、管理 peon 的 Middle Manager 组件和管理任务分配给 Middle Manager 的 Overlord 组件。其中 Middle Manager 和 peon 组件必须在同一个节点上。索引服务架构示意图如图 8-3 所示。



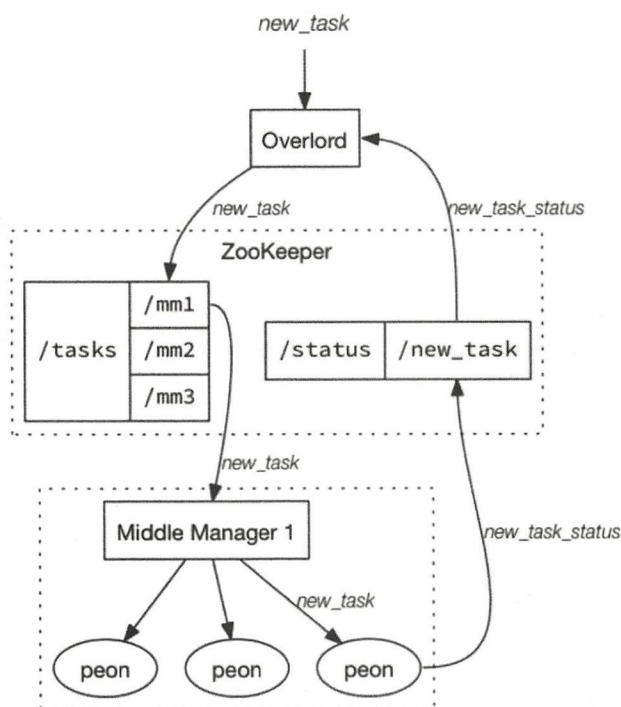


图8-4 索引服务架构示意图

2. Druid的外部依赖

除上面介绍的 5 个节点外，Druid 集群还有 3 个外部依赖。

(1) ZooKeeper 集群：用于集群服务的发现和当前数据拓扑的维护。

(2) Metadata Storage：用于存储数据文件的元数据，但是不存储实际的数据。可以使用 MySQL 或者 PostgreSQL 作为 Metadata Storage。Metadata Storage 一般会存储以下几个表。

- Segments Table：Segment 数据文件的元信息表。
- Rule Table：Segment 数据文件的存储规则表。
- Config Table：配置表，用于存储运行时的配置对象。
- Task-related Table：索引服务创建并使用到的表。
- Audit Table：审计表，用于存储配置变化的审计历史记录。



(3) Deep Storage: 用于存储 Segment 数据文件。

3. Segment传输过程

Segment 传输过程如图 8-5 所示。

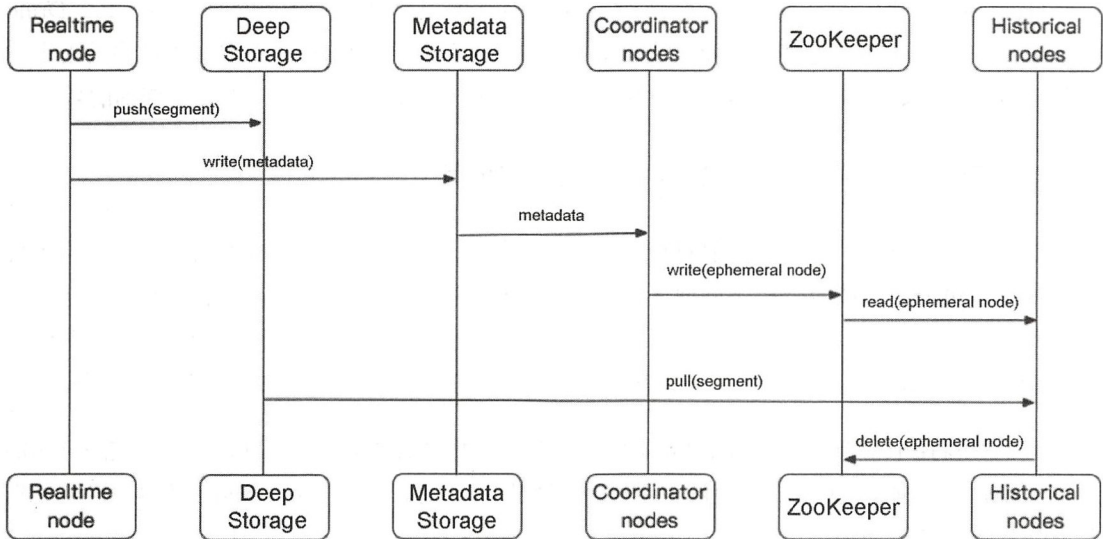


图8-5 Segment传输过程

实时数据写入后的 Segment 传输流程如下：

(1) 实时节点将在一定时间内收集到的数据生成一个 Segment 数据文件，并发送到 Deep Storage，同时将该 Segment 数据文件的元信息发送到元数据库中。

(2) 协调节点从元数据库中获取新的 Segment 数据文件的元信息，根据规则分配该 Segment 数据文件存储的历史节点，并将元数据写入在 ZooKeeper 上创建的一个临时节点中。

(3) 历史节点从 ZooKeeper 的临时节点中读取元数据，去 Deep Storage 中拉取指定的 Segment 数据文件，并删除在 ZooKeeper 上创建的临时节点。

4. DataSource

在 Druid 中，数据是存储在每个 DataSource 中的，而一个 DataSource 就相当于 MySQL 的一个总表。每个 DataSource 都会包含下面三部分信息。

○ **Timestamp:** 存储指标的 UTC 时间列，可以精确到毫秒。

- Dimension: 存储指标的维度列。

- Metric: 指标列, 用来聚合计算。

5. Segment

DataSource 描述的是数据的逻辑结构, 而 Segment 则体现了数据的物理存储方式。Druid 通过 segmentGranularity 参数来设置 Segment 划分的时间范围, 而 Druid 查询也正是通过指定的时间范围来确认需要查询的 Segment 数据文件的, 减小了查询的数据量, 提高了查询的效率。

8.3.3 Druid 在微博广告监控平台中的应用

在微博广告监控平台中, 需要对指标进行复杂的组合分析, 比如分析不同地区、不同年龄段下的请求量分布, 或者分析不同地区、不同用户标签下的请求量分布, 或者分析不同地区、不同场景流下的请求量分布。这样的多维多指标的组合同有上百上千种, 在这种场景下使用 Graphite 来分析显然是不合适的。所以我们引入了 Druid 来代替 Graphite 作为主数据引擎。

我们在监控平台架构中使用了 Flink 作为实时流数据的计算框架, 来实时消费存储在 Kafka 中的日志, 再根据一定的配置规则, 从日志中提取指标信息, 最后写入 Druid。微博广告监控指标数据提取示意图如图 8-6 所示。

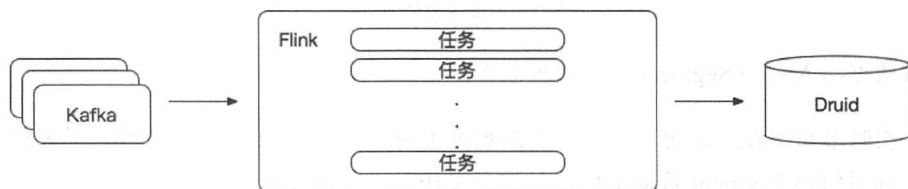


图8-6 微博广告监控指标数据提取示意图

所有的指标提取任务都是通过配置文件来设定提取规则的, 配置文件分为以下三部分。

- Input: 指定消费的 Topic 元信息。

- Filter: 日志清洗规则, 实现一套类似 Logstash 的过滤功能。

- Output: 指定写入的 Druid 元信息。

Output 的配置如下:

```
{
  "dataSources": [
```



```
{
  "spec": {
    "dataSchema": {
      "dataSource": "wb_ad_nginx",
      "parser": {
        "type": "string",
        "parseSpec": {
          "format": "json",
          "timestampSpec": {
            "column": "timestamp",
            "format": "posix"
          },
          "dimensionsSpec": {
            "dimensions": ["uri", "code", "host", "distributed"],
            "dimensionExclusions": [],
            "spatialDimensions": []
          }
        }
      },
    },
    "metricsSpec": [
      {
        "type": "doubleSum",
        "name": "cost_time",
        "fieldName": "cost_time"
      },
      {
        "type": "longSum",
        "name": "count",
        "fieldName": "count"
      }
    ],
    "granularitySpec": {
      "type": "uniform",
      "segmentGranularity": "HOUR",
      "queryGranularity": "SECOND"
    }
  },
  "tuningConfig": {
    "type": "realtime",

```

```
        "maxRowsInMemory": 100000,  
        "intermediatePersistPeriod": "PT10m",  
        "windowPeriod": "PT20M"  
    }  
},  
"properties" : {  
    "task.partitions" : "2",  
    "task.replicants" : "2"  
}  
}  
]  
}
```

下面分别介绍上面配置文件中各字段的含义。

(1) dataSchema

dataSource: 指定 Druid 中的数据源名称。

(2) parser

type: 声明解析一条数据的类型。

parseSpec: 定义数据格式、时间戳名称和维度的列表。

metricsSpec: 定义指标名称和指标聚合类型的列表。

granularitySpec: 指定每个 Segment 的存储粒度和最小查询粒度。

(3) tuningConfig——优化数据写入参数。

maxRowsInMemory: 在数据写入磁盘前内存存储的最大行数。

intermediatePersistPeriod: 多长时间数据临时写入磁盘一次。

windowPeriod: 时间窗口，超出时间的数据将被丢弃。

(4) properties

task.partitions: 定义分区数量。

task.replicants: 定义每个分区的副本数。

Druid 的数据聚合、多维分析、实时查询功能解决了监控平台的很多问题，但是在原始数据查询和高基数维度分析上，Druid 依然无能为力。所以我们又引入了 Elasticsearch 作为 Druid 的补充，填补 Druid 在原始数据查询和高基数维度分析方面的不足。但是在海量数据场景下，Elasticsearch 的查询性能依然不容乐观。同时两种数据引擎的维护成本也是非常高的，那么有没有一种数据引擎既能满足数据聚合、高基数维度分析、实时查询的需求，又能保留原始的数据呢？

8.4 性能神器 ClickHouse

8.4.1 什么是 ClickHouse

ClickHouse^[3]是 Yandex 开源的一款用于 OLAP 的高性能列式分析数据库。虽然 ClickHouse 是一个非常年轻的开源项目，但是它的性能却优于很多商业数据库，如 Vertica 等，更是秒杀其他的开源数据库，如 MySQL、Hive 等。图 8-7 展示了 ClickHouse 和其他数据库之间的性能对比情况。

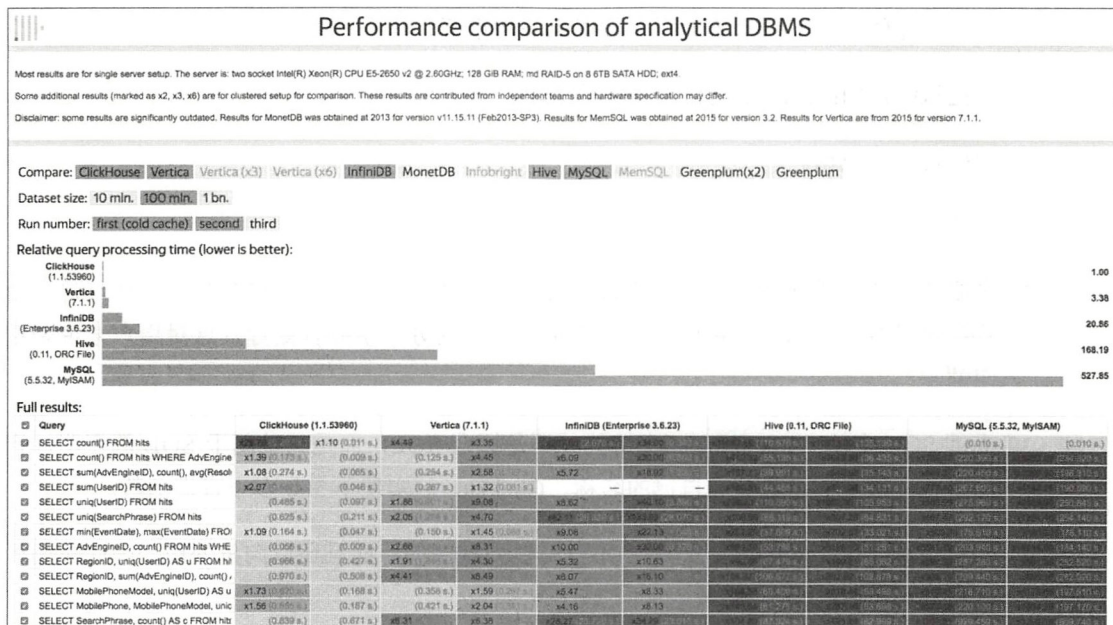


图8-7 数据库性能对比情况

Yandex 的统计分析服务 Yandex.Metrica，据说是世界上第二大网站分析平台。其通过近 400

台机器的 ClickHouse 集群，为每天 200 亿个事件和历史总记录达 13 万亿个事件做查询和分析，更重要的是这些事件都是以原始数据存储的，而不是聚合后的指标。

ClickHouse 为什么这么快？因为 ClickHouse 是用 C++ 语言编写的，除具有 C++ 本身的高性能外，根据 ClickHouse 的开发工程师所述，他们主要通过以下几点让 ClickHouse 可以这么快。

- Column Storage（列式存储结构，被设计成针对大数据的一种存储模式）
- LSM-Tree（Log-Structured Merge Tree，日志结构合并树）
- Vectorized Query Execution（向量化查询，一种高性能查询设计）
- Specialization（ClickHouse 针对特定查询的一系列优化）

8.4.2 ClickHouse 的特性

ClickHouse 具有如下特性。

- 真正面向列的 DBMS：在真正面向列的 DBMS 中，没有一起存储的“垃圾”，即使在没有压缩的情况下，紧凑地存储数据也非常重要。
- 数据压缩：ClickHouse 支持两种压缩方法，即 LZ4 和 ZSTD。
- 用磁盘存储数据。
- 多核并发处理。
- 多服务器分布式处理：在 ClickHouse 中，数据可以保存在不同的分片中，而且每个分片都可以有一组用于容错的副本。查询在所有的分片中并行处理，这对用户来说是透明的。
- 提供 SQL 支持：ClickHouse 支持的并不是标准 SQL，而是一种基于 SQL 的声明性查询语言，所有的函数都有自己的名称，ClickHouse 实现了 300 多个查询函数。
- 向量引擎（Vector Engine）：数据不仅是以列存储的，而且还以向量处理，因此我们可以充分利用 CPU 的性能。
- 实时数据更新：ClickHouse 支持主键表。为了快速地对主键的范围执行查询，可以使用 MergeTree 对数据进行增量排序。因此，可以不断地将数据添加到表中。添加数据

时没有锁。

- 支持索引。
- 支持设置主键。
- 支持在线查询。
- 支持近似计算：ClickHouse 包含了对各种值、中位数和分位数进行近似计算的聚合函数，在数据的样本集上执行查询，得到一个近似的结果。近似计算适合从磁盘检索比例较少的数据集。
- 数据复制和支持副本数据完整性：使用异步多主复制。在写入任何可用副本之后，数据被分发到所有剩余的副本上。系统在不同的副本上维护相同的数据。

8.4.3 ClickHouse 的不足

ClickHouse 的不足之处主要体现在如下几个方面。

- 不支持事务。
- 对于聚合，ClickHouse 查询的结果必须小于或等于单台服务器上的内存大小，不适合查询的结果数据量超出单节点内存的场景。
- 不支持更新、删除操作（后续版本会加上更新、删除的功能）。

8.4.4 安装配置 ClickHouse

1. 环境准备

由于 ClickHouse 官方倾向于 Ubuntu，所以只提供了 DEB 包和 Docker 镜像安装方式。但是由于我们的大多数生产环境都是 CentOS 系统，所以很感谢 Altinity 公司打包提供了 RPM 包，让我们摆脱了环境的问题。

（1）从 Altinity 官网下载 RPM 包

```
wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-client-1.1.54342-1.el7.x86_64.rpm/download.rpm
wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-debuginfo-1.1.54342-1.el7.x86_64.rpm/download.rpm
```

```
wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-server-1.1.54342-1.el7.x86_64.rpm/download.rpm
wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-server-common-1.1.54342-1.el7.x86_64.rpm/download.rpm
wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-test-1.1.54342-1.el7.x86_64.rpm/download.rpm
```

(2) 下载依赖包

```
wget ftp://www.rpmfind.net/linux/centos/7.4.1708/os/x86_64/Packages/unixODBC-2.3.1-11.el7.x86_64.rpm
```

(3) 安装 ClickHouse

```
rpm -ivh unixODBC-2.3.1-11.el7.x86_64.rpm
rpm -ivh clickhouse*
```

2. 配置单节点ClickHouse

通过 RPM 包安装后默认的配置文件的在 `/etc/clickhouse-server/` 目录下，可以通过修改 `/etc/init.d/clickhouse-server` 文件来改变配置文件的存放路径。Clickhouse 安装后默认生成两个配置文件：

```
# tree /etc/clickhouse-server/
/etc/clickhouse-server/
├─ config.xml
└─ users.xml
0 directories, 2 files
```

修改数据存储路径到数据分区：

```
# vi /etc/clickhouse-server/config.xml
<path>/data0/clickhouse/</path>
<tmp_path>/data0/clickhouse/tmp/</tmp_path>
```

3. 配置ClickHouse分布式集群

对于集群，可以在 `config.xml` 中进行配置，也可以在自定义的文件中进行配置。`config.xml` 默认包含了配置文件 `/etc/metrika.xml`，但是这个文件默认并不存在，需要手动创建。我们也可以调整 `metrika.xml` 配置文件的存放路径，然后在 `config.xml` 文件中通过 `include_from` 元素来设置新的存放路径，如将 `metrika.xml` 和 `config.xml` 放到同一个目录下。

```
# vi /etc/clickhouse-server/config.xml
<include_from>/etc/clickhouse-server/metrika.xml</include_from>
```

ClickHouse 通过 `metrika.xml` 配置文件中的 `clickhouse_remote_servers` 元素来指定集群配置，`config.xml` 将通过 `<remote_servers incl="clickhouse_remote_servers" />` 调用。

分片通过 `weight` 元素来指定接收数据的比例大小，如下例中，分片 1 的 `weight` 为 1，分片 2 的 `weight` 为 2，而实际数据写入时，分片 2 写入的量将是分片 1 的两倍。

```
<!-- 以 config.xml 中指定的元素名称开始 -->
<clickhouse_remote_servers>
  <!-- 集群名称 -->
  <ck_cluster>

    <!-- 数据分片 1 -->
    <shard>
      <!-- 配置写入数据分片 1 的权重 -->
      <weight>1</weight>
      <!-- 复制分片的配置 -->
      <internal_replication>false</internal_replication>
      <replica>
        <host>192.168.1.1</host>
        <port>9000</port>
        <user>default</user>
        <password>123456</password>
      </replica>
    </shard>

    <!-- 数据分片 2 -->
    <shard>
      <!-- 配置写入数据分片 2 的权重 -->
      <weight>2</weight>
      <internal_replication>false</internal_replication>
      <replica>
        <host>192.168.1.2</host>
        <port>9000</port>
        <user>default</user>
        <password>123456</password>
      </replica>
    </shard>

  </ck_cluster>
```

```
<!-- 以 config.xml 中指定的元素名称结束 -->
</clickhouse_remote_servers>
```

关于分布式表的创建，详见 8.4.5 节。

4. 配置 ClickHouse 高可用集群

复制工作在单个表上，而不是整个服务中，所以在一个 ClickHouse 服务中可以同时存在副本表和非副本表。副本与分片无关，副本独立工作于每个分片中。

- 可以被复制的语句：INSERT, ALTER, Compressed data。
- 不能被复制的语句：CREATE, DROP, ATTACH, DETACH, RENAME。

复制是异步进行的，如果部分副本不可用，数据将在它们恢复可用后再写入。如果副本可靠，延迟时间就是网络传输压缩数据块的时间。

ClickHouse 通过 ZooKeeper(3.4.5 或以上版本)在多个副本间复制数据，且只支持 MergeTree Family 表引擎。

配置示例如下：

```
<!-- ZooKeeper 配置 -->
<zookeeper-servers>
  <node index="1">
    <host>192.168.11.1</host>
    <port>2181</port>
  </node>
</zookeeper-servers>
```

查询操作不经过 ZooKeeper，所以复制操作不会影响查询性能。当查询分布式复制表时，可以通过下面两个参数来控制 ClickHouse 的查询行为。

- `max_replica_delay_for_distributed_queries`：在进行分布式查询时，如果副本超过了所设置的时间，那么将不会被使用。
- `fallback_to_stale_replicas_for_distributed_queries`：ClickHouse 从表的过时副本中选择最相关的内容

由于每个复制表在 ZooKeeper 中的路径必须不一样，同一个复制表在不同的分片中也需要有不一样的路径，所以我们可以通过宏定义动态生成 ZooKeeper 路径，如上面的 {shard} 和 {replica} 都是配置文件中定义的宏，这样在创建表时，就可以根据实例所在主机在 ZooKeeper 中动态设

置对应的路径和副本名称。

配置示例如下：

```
<!-- 定义宏 -->
<macros>
  <shard>01</shard>
  <replica>192.168.1.1</replica>
</macros>
```

关于分布式复制表的创建，详见 8.4.5 节。

5. 启动ClickHouse

```
service clickhouse-server start
```

6. 连接ClickHouse

```
clickhouse-client -h 192.168.1.1 -m -u default --password 123456
```

8.4.5 表引擎

1. MergeTree表引擎

MergeTree 表引擎是 ClickHouse MergeTree Family 的一员，也是 ClickHouse 中最主要的表引擎。MergeTree 表引擎支持按主键和 Date 索引。

MergeTree 表引擎至少需要三个参数：

- 包含日期的日期类型列。
- 主键列表。
- 索引粒度。

配置示例如下：

```
MergeTree(EventDate, (CounterID, EventDate), 8192)
```

如果对数据采样，还需要第四个参数：采样表达式，那么配置就是这样的：

```
MergeTree(EventDate, intHash32(UserID), (CounterID, EventDate, intHash32(UserID)), 8192)
```

在 MergeTree 表类型的表中，必须包含 date 列。在上面的例子中，EventDate 就是 date 列，date 列的类型必须是 Date 类型。

采样表达式可以是任何表达式，但是必须是主键之一。上例中使用用户 id 的 hash 值作为

采样列。

完整的 MergeTree 表引擎配置示例如下：

```
CREATE TABLE IF NOT EXISTS default.nginx ON CLUSTER ck_cluster
(
    date Date,
    datetime DateTime,
    code String,
    cost_time UInt16
) ENGINE = MergeTree(date, (date, datetime), 8192)
```

在建表语句中添加“ON CLUSTER 集群名称”，可以在集群的所有机器上一键建表。

默认 ClickHouse 以月划分分区，这样导致如果想删除历史数据也只能按月删除。从 1.1.54310 版本开始，ClickHouse 支持自定义分区，那么我们可以定义一天一个分区，这样就可以按天删除历史数据了。

○ PARTITION BY：分区表达式。

○ ORDER BY：主键列表。

○ SAMPLE BY：采样表达式。

MergeTree 表引擎自定义分区配置示例如下：

```
CREATE TABLE IF NOT EXISTS default.nginx ON CLUSTER ck_cluster
(
    date Date,
    datetime DateTime,
    code String,
    cost_time UInt16
) ENGINE = MergeTree PARTITION BY date ORDER BY (date, datetime) SETTINGS index_granularity
= 8192;
```

按天删除历史数据：

```
ALTER TABLE default.nginx DROP PARTITION ('2018-02-14');
```

2. Distributed表引擎

通过 MergeTree 表引擎创建的表只作用于单台机器，那么如何把每个节点上的 MergeTree 表关联起来，形成一个分布式表呢？

ClickHouse 通过 Distributed 表引擎来关联不同节点上的 MergeTree 表,从而形成分布式表。Distributed 表本身并不存储数据,它只是关联本机上面的 MergeTree 表。因此,我们在创建分布式表前,必须在每个节点上创建 MergeTree Family 表。

Distributed 表允许在多台服务器上进行分布式写入和查询处理。分布式引擎需要以下几个参数:

- 集群名称
- 数据库的名称
- 关联的表名称
- 分区键值 (Sharding key)

配置示例如下:

```
Distributed(ck_cluster, 'default', 'nginx', rand())
```

通过 Distributed 表查询,数据会在每个节点上进行聚合,然后将结果集发送给请求的节点,请求节点再对各个节点上的结果集进行聚合,最后得到最终的结果。

通过 Distributed 表写入事件时,数据会通过 Distributed 表中 Sharding key 指定的方法来分发事件。我们默认使用 rand()随机分发事件到每个节点上,或者通过 Nginx 请求的 code 来分发事件,那么相同 code 的事件将写到同一个节点上。但是 Nginx 的 code 一般都是不均匀分布的,直接按照 code 分发,节点之间的数据就会不平衡,我们可以通过对指定字段 hash 来让事件分布均匀,如 intHash64(code)。配置示例如下:

```
CREATE TABLE IF NOT EXISTS default.nginx_all ON CLUSTER ck_cluster
(
    date Date,
    datetime DateTime,
    code String,
    cost_time UInt16
) ENGINE = Distributed(ck_cluster, 'default', 'nginx', rand());
```

3. ReplicatedMergeTree表引擎

创建了本地表和分布式表,但是它们都没有副本,没有实现高可用性,这就意味着如果有一个节点宕机了,那么这个节点上的数据都将不可用。

副本表通过 ZooKeeper 同步数据，上面在对 ClickHouse 集群进行配置时，已经配置了 ZooKeeper 和宏定义，现在我们只要用副本表取代 MergeTree 表就可以实现数据副本。副本表引擎除需要提供 MergeTree 表所需的几个参数外，还需要指定 ZooKeeper 的节点路径和名称两个参数。

```
ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/hits', '{replica}', EventDate,
intHash32(UserID), (CounterID, EventDate, intHash32(UserID), EventTime), 8192)
```

完整的副本表配置示例如下：

```
CREATE TABLE IF NOT EXISTS default.nginx ON CLUSTER ck_cluster
(
    date Date,
    datetime DateTime,
    code String,
    cost_time UInt16
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/nginx', '{replica}')
PARTITION BY date ORDER BY (date, datetime) SETTINGS index_granularity = 8192;
```

在副本表的基础上创建分布式表（配置见“Distributed 表引擎”一节），就可以实现一个分布式副本表。

4. AggregatingMergeTree表引擎

在已有的分布式副本表上，我们还可以对其创建物化聚合视图表，预聚合我们要查询的数据，提高查询的响应速度。

配置示例如下：

```
CREATE TABLE IF NOT EXISTS default.nginx ON CLUSTER ck_cluster
(
    date Date,
    datetime DateTime,
    code String,
    cost_time UInt16
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/nginx', '{replica}')
PARTITION BY date ORDER BY (date, datetime) SETTINGS index_granularity = 8192;
```

除了上面介绍的这些表引擎，ClickHouse 还提供了很多表引擎，比如直接从 Kafka 读取事件的表引擎、Memory 表引擎等。想了解更多内容，请参考官方文档：https://clickhouse.yandex/docs/en/table_engines。

8.4.6 函数支持

ClickHouse 官方实现了 300 多个函数支持实时的数据查询与分析。ClickHouse 将函数分为三种类型：常规函数、聚合函数和 ARRAY JOIN。下面分别列出各种类型下的一些常用函数。

1. 常规函数

常规函数作用于某一行，其结果不受其他行的影响。

(1) 算术函数

- `plus(a, b)`: $a+b$
- `minus(a, b)`: $a-b$
- `multiply(a, b)`: $a * b$
- `divide(a, b)`: a / b , 结果为浮点型, 并非整除
- `intDiv(a, b)`: 整除, b 不能为 0
- `intDivOrZero(a, b)`: 整除, b 可以为 0。若 b 为 0, 则结果等于 0
- `modulo(a, b)`: $a \% b$
- `negate(a)`: $-a$
- `abs(a)`: 取绝对值
- `gcd(a, b)`: 取 a 与 b 的最大公因数
- `lcm(a, b)`: 取 a 与 b 的最小公倍数

(2) 类型转换函数

- `toUInt8`
- `toInt8`
- `toFloat32`
- `toUInt8OrZero`
- `toDate`: 2018-01-01
- `toDateTime`: 2018-01-01 01:00:00

- toString
- cast(x,t): 将 x 类型转换成 t 类型

(3) 日期函数

- toHour: 找出指定时间所在的小时
- toStartOfHour: 找出从指定时间开始的小时
- toRelativeHourNum: 统计从某个固定点开始到现在的小时数
- now: 返回当前时间, 如 2018-01-01 01:01:01
- today: 返回当前日期, 如 2018-01-01

2. 聚合函数

- count(): 统计行数
- max(): 统计最大值
- sum(): 计算和
- uniq(): 计算不同值的数量
- topK(N)(column): 计算指定列中出现次数最多的 N 个值
- runningDifference(): 计算指定列相邻值的差
- quantile(level)(x): 计算指定列水平分位数, level 是一个 0~1 的浮点数, 比如可以用它计算 95% 的耗时分布
- groupUniqArray(x): 返回指定列 x 出现的不同值, 相当于 SQL 中的 distinct
- uniqUpTo(N)(x): 统计指定列 x 出现不同值的个数, 相当于对 groupUniqArray 的结果做 count。如果返回的值小于指定的 N , 则显示实际的值; 如果返回的值大于 N , 则显示 $N+1$ 个数
- -If: 如果函数以 If 为后缀, 如 CountIf(), 这时函数将接收一个额外的参数, 用来判断是否满足条件
- -Array: 函数以 Array 为后缀, 如 sumArray(arr), 计算数组 arr 中元素的和

3. ARRAY JOIN

普通函数只会改变每行中的值，聚合函数会压缩一组行，而 ARRAY JOIN 则是将一行展开成多行。该函数以数组作为参数。示例如下：

(1) 创建测试表

```
CREATE TABLE test
(
    s String,
    arr Array(String)
)
ENGINE = Memory;
```

(2) 导入测试数据

```
INSERT INTO test VALUES ('test1', ['11', '12', '13', '11']), ('test2', ['a', 'b', 'c', 'd']);
```

(3) 返回 ARRAY JOIN

```
SELECT
    s,
    arr,
    a
FROM test
ARRAY JOIN arr AS a;
```

(4) 执行结果

执行结果如图 8-8 所示。

s	arr	a
test1	['11', '12', '13', '11']	11
test1	['11', '12', '13', '11']	12
test1	['11', '12', '13', '11']	13
test1	['11', '12', '13', '11']	11
test2	['a', 'b', 'c', 'd']	a
test2	['a', 'b', 'c', 'd']	b
test2	['a', 'b', 'c', 'd']	c
test2	['a', 'b', 'c', 'd']	d

图8-8 ARRAY JOIN执行结果

欲进一步了解更多的函数及其使用方法，读者可以查看 ClickHouse 官方文档^[4]。

8.5 本章小结

从 Graphite 到 Druid 再到 ClickHouse，我们发现虽然监控需求的深度和广度越来越大，但是数据引擎的发展也是非常迅速的，像 Druid、ClickHouse 这样的专为海量数据多维分析而生的数据引擎，给大数据时代下的运维带来了极大的便利。当面对一个复杂的系统时，我们依然可以从容地展示系统内部各个环节的性能和业务指标。

8.6 参考文献

[1] <https://graphite.readthedocs.io/>

[2] <http://druid.io/docs/0.12.0/design/index.html>

[3] <https://clickhouse.yandex/docs/en>

[4] <https://clickhouse.yandex/docs/en/functions>

第9章

机器学习框架

9.1 简介

常见的机器学习框架有 Theano、TensorFlow、Torch、Caffe、MXNet、Neon 和 CNTK 等，每个框架都有自己的特点，比如 Caffe 支持 C++，对 CNN 的支持非常好；Theano 的文档非常详细；TensorFlow 支持更大的并发处理和复杂的模型等。图 9-1 列举了几个常见的机器学习框架的对比情况。

	Caffe	Torch	Theano	TensorFlow
Language	C++, Python	Lua	Python	Python
Pretrained	Yes ++	Yes ++	Yes (Lasagne)	Inception
Multi-GPU: Data parallel	Yes	Yes cunn. DataParallelTable	Yes platoon	Yes
Multi-GPU: Model parallel	No	Yes fbcunn.ModelParallel	Experimental	Yes (best)
Readable source code	Yes (C++)	Yes (Lua)	No	No
Good at RNN	No	Mediocre	Yes	Yes (best)

图9-1 常见的机器学习框架的对比情况

（图片来源：斯坦福的CS231n课件——*Convolutional Neural Networks for Visual Recognition(Winter 2016)*）

读者可以结合业务并根据这些框架的特点进行选择。TensorFlow 作为 Google 的开源机器学习框架，开发者的使用规模比较大，文档较为丰富，非常方便入门学习。同时，TensorFlow 集成了很多算法模型案例，极大地降低了学习成本。接下来我们将重点介绍 TensorFlow 的安装使

用，以及如何进行模型训练。

9.2 TensorFlow介绍

9.2.1 什么是 TensorFlow

TensorFlow^[1]是一个采用数据流图（Data Flow Graph），用于数值计算的开源软件库。节点（Node）在图中表示数学操作，图中的线（Edge）则表示在节点间相互联系的多维数据数组，即张量（Tensor）。TensorFlow 最初是由 Google 大脑小组（隶属于 Google 机器智能研究机构）的研究员和工程师开发出来的，用于机器学习和深度学习方面的研究，但这个系统的通用性使其也可被广泛用于其他计算领域。其灵活的架构支持在多种平台上展开计算，例如台式计算机中的一个或多个 CPU（或 GPU）、服务器、移动设备等。

9.2.2 下载安装

读者可以在 TensorFlow 的官网下载二进制包，或者使用源码进行安装，最新的 TensorFlow 官方版本支持的操作系统（64 位）环境为：

- Mac OS X 10.11 (El Capitan) 及以上版本
- Ubuntu 16.04 及以上版本
- Windows 7 及以上版本

TensorFlow 不仅支持不同的操作系统，同时还支持 Python、Java、C 和 Go 等编程语言环境，降低了使用门槛。不同的操作系统及编程语言环境的安装流程略有不同，本文将以 Python 和 Ubuntu 操作系统为例进行安装说明。对于 Python 编程环境，官方推荐使用基于 VirtualEnv 的安装方式。

1. 安装Python及VirtualEnv

在 Python 官方网站根据不同的操作系统下载源码包（python2.7.11），如图 9-2 所示。Ubuntu 系统可以选择下载 xz 源码包（注意：对于 TensorFlow 的 Python 版本需要大于 2.7，当然也可以安装 Python 3.0 及以后的版本，此时 pip 的版本会略有不同）。

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		6b6076ec9e93f05dd63e47eb9c15728b	16856409	SIG
XZ compressed source tarball	Source release		1dbcc848b4cd8399a8199d000f9f823c	12277476	SIG
Mac OS X 32-bit i386/PPC installer	Mac OS X	for Mac OS X 10.5 and later	8d563a63b261fc3868c101471442b601	24018001	SIG
Mac OS X 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	cacd8b6a05c5a5c0f0e19f684a0c7f10	22162527	SIG
Windows debug information files	Windows		b5ebe6703d69ee97d1d648d20df6ee55	24359078	SIG
Windows debug information files for 64-bit binaries	Windows		34b3e9342b7a9dd58e0f20c6108e72e6	25104550	SIG
Windows help file	Windows		0d8044f1da197c8381be0789c2d5cc98	6171837	SIG
Windows x86-64 MSI installer	Windows	for AMD64/EM64T/x64	25acca42662d4b02682eee0df3f3446d	19550208	SIG
Windows x86 MSI installer	Windows		241bf8e097ab4e1047d9bb4f59602095	18636800	SIG

图9-2 Python下载页面

然后解压源码：

```
tar -xvf Python-2.7.11.tgz
$ tar -xvf Python-2.7.11.tgz # 解压下载的 Python 2.7.11.tgz 压缩包
```

执行编译安装。注意：该操作可能需要具有 root 权限（或者 sudo 执行）。

```
$ ./configure # 配置生成 Makefile
$ make # 编译
$ make install # 安装到系统中
```

完成后，可以在 shell 下执行 python 命令查看是否安装成功，以及 Python 的具体版本信息。接下来更新安装 pip，pip 是 Python 的包管理工具，非常方便安装和管理 Python 的各种依赖包。安装 Python 2.7.11 后默认已经安装好 pip，如果没有安装，则可以通过以下命令进行安装。

```
$ apt-get install python-pip # 安装 pip
```

通过以下命令进行 pip 的自身升级。

```
$ pip install -U pip # pip 的升级更新
```

VirtualEnv 通过创建一个虚拟化的 Python 运行环境，将我们所需的依赖安装进去，形成一个隔离的虚拟环境，不同的项目之间互不干扰，解决了依赖冲突的问题。

```
$ sudo apt-get install python-dev python-virtualenv # for Python 2.7
```

创建一个虚拟环境：

```
# 创建虚拟环境，并设置路径为当前用户根目录下的 tensorflow 路径
$ virtualenv --system-site-packages ~/tensorflow
```

执行下面的命令激活

```
$ source ~/tensorflow/bin/activate # bash, sh, ksh, or zsh
```

激活成功后路径会被修改成

```
(tensorflow)$
```

升级更新 pip 的版本，并将 TensorFlow 安装到当前虚拟环境中。

```
(tensorflow)$ easy_install -U pip
```

```
(tensorflow)$ pip install --upgrade tensorflow
```

注意：开启 GPU 版本的支持，请使用下面的命令进行安装

```
#(tensorflow)$ pip install --upgrade tensorflow-gpu
```

完成后，通过下面的命令退出 VirtualEnv，shell 提示符恢复原状。

当使用完 TensorFlow 时，解除激活

```
(tensorflow)$ deactivate # 停用 VirtualEnv
```

安装完成之后，可以执行 `tf.__version__` 命令查看 TensorFlow 的版本号，如图 9-3 所示。

```
(tensorflow) AndrewdeMacBook-Pro:tensorflow pengdong3$ python
Python 2.7.11 (default, Jan 22 2016, 08:29:18)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> tf.__version__
'1.7.0'
>>> tf.__path__
['/Users/andrew/tensorflow/lib/python2.7/site-packages/tensorflow']
>>>
```

图9-3 查看TensorFlow的版本号

至此，通过 VirtualEnv 方式就安装好了 TensorFlow。下一节我们将通过一个简单的例子来使用 TensorFlow。

2. Docker安装

Docker 安装方式能够快速构建 TensorFlow 的运行环境，执行以下命令即可。

方法一：使用 Docker 安装，启动一个具有 TensorFlow 及依赖环境的容器

b.gcr.io 为 Google 服务器，读者需要保证网络能够访问该域名

```
$ docker run -it b.gcr.io/tensorflow/tensorflow
```

方法二：安装具有 TensorFlow 源码版本的镜像

```
$ docker run -it b.gcr.io/tensorflow/tensorflow-full
```

方法三：启动具有 Jupyter 工具的 TensorFlow 镜像

此时，可以通过 `http://localhost:8888` 访问 Jupyter 服务，进行在线编辑和执行


```
# Python 代码
$ docker run -it -p 8888:8888 tensorflow/tensorflow
```

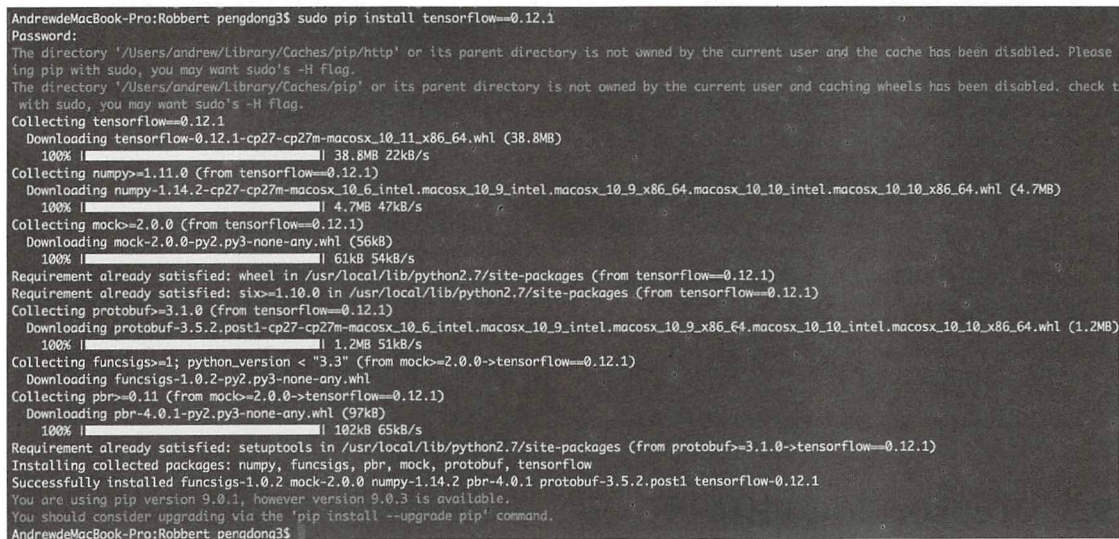
其中,通过第三种方法启动的镜像,会直接安装好 Jupyter Notebook 服务,通过 Web 浏览器可以对代码进行编辑,同时也能够运行 shell,非常方便。

3. pip安装

通过 pip 安装相对简单,只需要安装好 pip 工具,执行下面的命令即可。

```
# pip 安装最新版本
$ pip install tensorflow
# pip 安装指定版本
$ pip install tensorflow==0.12.1
# pip 安装指定版本, GPU 版本
$ pip install tensorflow-gpu==0.12.1
```

安装过程如图 9-4 所示。



```
AndrewdeMacBook-Pro:Robbert pengdong3$ sudo pip install tensorflow==0.12.1
Password:
The directory '/Users/andrew/Library/Caches/pip/http' or its parent directory is not owned by the current user and the cache has been disabled. Please
ing pip with sudo, you may want sudo's -H flag.
The directory '/Users/andrew/Library/Caches/pip' or its parent directory is not owned by the current user and caching wheels has been disabled. check t
with sudo, you may want sudo's -H flag.
Collecting tensorflow==0.12.1
  Downloading tensorflow-0.12.1-cp27-cp27m-macosx_10_11_x86_64.whl (38.8MB)
    100% |#####| 38.8MB 22kB/s
Collecting numpy>=1.11.0 (from tensorflow==0.12.1)
  Downloading numpy-1.14.2-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl (4.7MB)
    100% |#####| 4.7MB 47kB/s
Collecting mock>=2.0.0 (from tensorflow==0.12.1)
  Downloading mock-2.0.0-py3-none-any.whl (56kB)
    100% |#####| 61kB 54kB/s
Requirement already satisfied: wheel in /usr/local/lib/python2.7/site-packages (from tensorflow==0.12.1)
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python2.7/site-packages (from tensorflow==0.12.1)
Collecting protobuf>=3.1.0 (from tensorflow==0.12.1)
  Downloading protobuf-3.5.2.post1-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl (1.2MB)
    100% |#####| 1.2MB 51kB/s
Collecting funcsigs>=1; python_version < "3.3" (from mock>=2.0.0->tensorflow==0.12.1)
  Downloading funcsigs-1.0.2-py2.py3-none-any.whl
Collecting pbr>=0.11 (from mock>=2.0.0->tensorflow==0.12.1)
  Downloading pbr-4.0.1-py2.py3-none-any.whl (97kB)
    100% |#####| 102kB 65kB/s
Requirement already satisfied: setuptools in /usr/local/lib/python2.7/site-packages (from protobuf>=3.1.0->tensorflow==0.12.1)
Installing collected packages: numpy, funcsigs, pbr, mock, protobuf, tensorflow
Successfully installed funcsigs-1.0.2 mock-2.0.0 numpy-1.14.2 pbr-4.0.1 protobuf-3.5.2.post1 tensorflow-0.12.1
You are using pip version 9.0.1, however version 9.0.3 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
AndrewdeMacBook-Pro:Robbert pengdong3$
```

图9-4 通过pip安装指定版本的TensorFlow

注意: 如果使用 Python 3.*版本,则需要配合 pip 3 来安装不同版本的 TensorFlow。因为安装后的 tensorflow 目录会被放置到对应 Python 版本的安装路径下(如图 9-3 所示, TensorFlow 被安装到 python2.7 路径下),比如使用 pip 3 安装 TensorFlow 最新版本,可以执行以下命令:

```
# pip 3 安装最新版本
$ pip3 install tensorflow
```

9.2.3 “Hello TensorFlow” 示例

在 VirtualEnv 中运行 python 命令，即可以通过 Python API 来使用 TensorFlow。下面展示了打印 “Hello TensorFlow” 的代码示例。

```
(tensorflow)$ python
Python 2.7.11 (default, Jan 22 2016, 08:29:18)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> hello = tf.constant('Hello TensorFlow')
>>> sess = tf.Session()
>>> print sess.run(hello)
Hello TensorFlow
```

注意：如果在 Mac 环境下通过 pip 安装，则可能会遇到以下警告信息，通常是因为没有按照操作系统 CPU 类型进行编译造成的，可以下载源码进行编译安装。

```
2018-04-02 21:34:15.140241: I
tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports instructions that
this TensorFlow binary was not compiled to use: AVX2 FMA
```

当然，也可以使用下面的命令忽略此条警告。

```
>>> Import os
# log 级别设置，取值 1,2,3，级别递增
# 设置为 1，显示所有信息（默认设置）
# 设置为 2，显示 Error 和 Warning
# 设置为 3，只显示 Error
>>> Os.environ[TF_CPP_MIN_LOG_LEVEL] = '2'
```

9.3 TensorFlow进阶

TensorFlow 官方提供了不少模型 demo 可供参考，比如手写数字识别的 MNIST 模型，如图 9-5 所示。

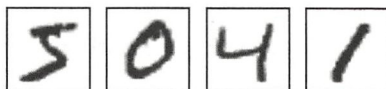


图9-5 MNIST手写体

首先，执行以下命令下载官方源码。

```
$ git clone https://github.com/tensorflow/tensorflow.git
```

然后，在源码树的根路径下执行下面的命令，运行 MNIST 的演示程序。

```
$ cd tensorflow/models/image/mnist
$ python convolutional.py
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting data/train-images-idx3-ubyte.gz
Extracting data/train-labels-idx1-ubyte.gz
Extracting data/t10k-images-idx3-ubyte.gz
Extracting data/t10k-labels-idx1-ubyte.gz
Initialized!
Epoch 0.00
Minibatch loss: 12.054, learning rate: 0.010000
Minibatch error: 90.6%
Validation error: 84.6%
Epoch 0.12
Minibatch loss: 3.285, learning rate: 0.010000
Minibatch error: 6.2%
Validation error: 7.0%
...
...
```

MNIST 的源码及使用说明可以参考官方源码库及相关文档，本节我们将使用 TensorFlow 来实现一个有意思的 NLP（自然语言处理）模型——机器自动生成诗词或短文。

9.3.1 基础理论

下面我们来介绍 RNN 最重要的一个变种：seq2seq 模型，又叫 Encoder-Decoder 模型。Encode 的意思是将输入序列转换成一个固定长度的向量，Decode 的意思是将输入的固定长度的向量解码成输出序列。其中编码/解码的方式可以是 RNN、CNN，Encode 是一个 RNNCell（RNN、GRU、LSTM 等）结构^[2]，如图 9-6 所示。

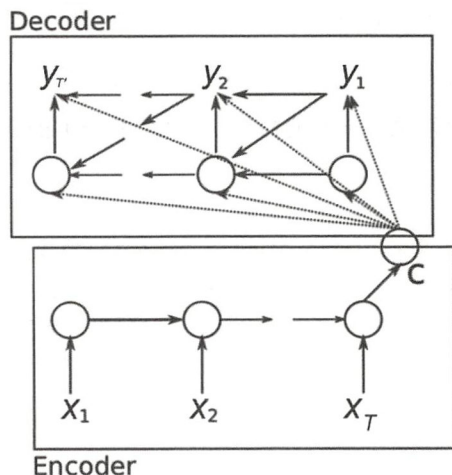


图9-6 seq2seq中的Encoder-Decoder结构

(图片来源：见参考文献[2])

关于 RNN 模型，可以参阅本书第 13 章中的相关内容。

关于 seq2seq 模型的具体说明，读者可以参阅相关文档。

原始的 RNN 要求序列等长，然而，我们遇到的大部分问题序列都是不等长的，比如在机器翻译中，源语言和目标语言的句子往往并没有相同的长度。Google 就基于 seq2seq 开发了一个对话模型^[3]，使用两个 LSTM 的结构，其中 LSTM1 将输入的对话编码成一个固定长度的实数向量；LSTM2 根据这个向量不停地预测后面的输出（解码）。只是在对话模型中，使用的语料是 ((input) 你说的话-我答的话 (input)) 这种类型的 pairs；而在机器翻译中，使用的语料是 (hello-你好) 这样的 pairs。

由于这种 Encoder-Decoder 结构不限制输入和输出的序列长度，因此应用的范围非常广泛。比如：

- 机器翻译——Encoder-Decoder 结构的最经典应用，事实上，这一结构就是在机器翻译领域最先提出的。
- 文本摘要——输入是一个文本序列，输出是这个文本序列的摘要序列。
- 阅读理解——对输入的文章和问题分别编码，再对其进行解码得到问题的答案。
- 语音识别——输入是语音信号序列，输出是文字序列。

9.3.2 模型准备

引入 TensorFlow 的相关包，这里需要使用 ops 下的 rnn_cell 和 seq2seq 包，代码如下：

```
#coding=utf-8
import os
import sys
import time

import numpy as np
import tensorflow as tf
from tensorflow.contrib.tensorboard.plugins import projector
from tensorflow.python.ops import rnn_cell, seq2seq
```

对于 rnn 和 seq2seq 模型，需要用到的参数可以通过 loadParam 函数进行加载配置，如批处理的大小（batch_size）、迭代次数等。部分参数的加载配置如下：

```
class loadParam():

    batch_size = 32 # 批处理大小
    n_epoch = 100 # 迭代次数
    learning_rate = 0.01
    decay_steps = 1000
    decay_rate = 0.9 # 配置衰减率
    grad_clip = 5

    state_size = 100
    num_layers = 3 # RNN 深度
    seq_length = 20
    log_dir = './logs'
    metadata = 'metadata.tsv'
    gen_num = 50 # 生成句子/文章摘要的字符数
```

9.3.3 训练数据

接下来准备训练的样本数据，这里摘录了新浪新闻最近 1000 余篇新闻报道作为训练数据集。为了保证数据编码的一致性，通常需要进行 UTF-8 编码，对加载后的数据进行预处理后保存到 meta 文件中。

```
class loadDataset():
```

```

def __init__(self, datafiles, args):
    self.seq_length = args.seq_length
    self.batch_size = args.batch_size
    # 打开文件以 UTF-8 进行统一编码
    with open(datafiles, encoding='utf-8') as f:
        self.data = f.read()

    # 保存基础信息
    self.total_len = len(self.data)
    self.words = list(set(self.data))
    self.words.sort()

    self.vocab_size = len(self.words)
    print('vocabulary size: ', self.vocab_size)
    self.char2id_dict = {w: i for i, w in enumerate(self.words)}
    self.id2char_dict = {i: w for i, w in enumerate(self.words)}

    # 批处理的指针位置，初始化为 0（开始位置）
    self._pointer = 0

    # 保存 metadata 信息
    self.save_metadata(args.metadata)

```

其中 meta 信息记录了所有 id 和 char 字符的映射关系。save_metadata 函数如下：

```

def save_metadata(self, file):
    with open(file, 'w') as f:
        f.write('id\tchar\n')
        for i in range(self.vocab_size):
            c = self.id2char(i)
            if c>="\u4e00" and c<="\u9fa5":
                f.write('{}\t{}\n'.format(i, c))

```

处理 id 和 char 字符映射关系的函数如下：

```

# 映射关系：char 到 id
def char2id(self, c):
    return self.char2id_dict[c]

# 映射关系：id 到 char
def id2char(self, id):
    return self.id2char_dict[id]

```

同时，需要定义在模型训练中每次加载的模型训练数据，`next_batch` 函数提供了对训练数据的访问功能，如下所示。

每次迭代都需要更新迭代器，获取到当前批处理需要的数据

```
def next_batch(self):
    x_batches = []
    y_batches = []
    for i in range(self.batch_size):
        if self._pointer + self.seq_length + 1 >= self.total_len:
            self._pointer = 0
        bx = self.data[self._pointer: self._pointer + self.seq_length]
        by = self.data[self._pointer +
                        1: self._pointer + self.seq_length + 1]
        # 更新迭代器的指针
        self._pointer += self.seq_length

        bx = [self.char2id(c) for c in bx]
        by = [self.char2id(c) for c in by]
        x_batches.append(bx)
        y_batches.append(by)

    return x_batches, y_batches
```

至此，模型所需要的数据集加载和处理，以及相关参数配置就全部完成了。

9.3.4 模型训练

模型训练主要包含模型选择定义（Definite Cell）、损失函数（Loss Function）定义、模型优化（Optimize）和模型迭代训练（Train）四个主要部分。

1. 模型选择定义

TensorFlow 提供的 RNN Cell 共有三种，分别是 RNN、GRU 和 LSTM，并且每一种都可以使用多层结构，即 MultiRNNCell，此处以 LSTM 为例进行说明。

```
with tf.name_scope('model'):
```

```
    self.cell = rnn_cell.BasicLSTMCell(args.state_size)
    self.cell = rnn_cell.MultiRNNCell([self.cell] * args.num_layers)
    # 初始化
```

```

self.initial_state = self.cell.zero_state(
    args.batch_size, tf.float32)
with tf.variable_scope('rnnlm'):
    w = tf.get_variable(
        'softmax_w', [args.state_size, data.vocab_size])
    b = tf.get_variable('softmax_b', [data.vocab_size])
    with tf.device("/cpu:0"):
        embedding = tf.get_variable(
            'embedding', [data.vocab_size, args.state_size])
        inputs = tf.nn.embedding_lookup(embedding, self.input_data)
    outputs, last_state = tf.nn.dynamic_rnn(
        self.cell, inputs, initial_state=self.initial_state)

```

2. 定义损失函数

TensorFlow 提供了 `sequence_loss_by_example` 函数用于按照权重来计算整个序列中每个单词的交叉熵，返回的是每个序列的 `log-perplexity`。`sequence_loss_by_example` 函数有三个参数，其中第一个参数表示通过 `matmul` 函数计算 `output` 和 `w` 矩阵相乘再与 `b` 进行矩阵求和；第二个参数是目标值矩阵；第三个参数为权重（`weight`）。`sequence_loss_by_example` 函数的返回值即为 `loss` 值。

定义损失函数

```

with tf.name_scope('loss'):
    output = tf.reshape(outputs, [-1, args.state_size])

    self.logits = tf.matmul(output, w) + b
    self.probs = tf.nn.softmax(self.logits)
    self.last_state = last_state

    targets = tf.reshape(self.target_data, [-1])

    #使用 seq2seq 提供的 sequence_loss_by_example 函数
    loss = seq2seq.sequence_loss_by_example([self.logits],
        [targets], [tf.ones_like(targets, dtype=tf.float32)])

    self.cost = tf.reduce_sum(loss) / args.batch_size
    tf.scalar_summary('loss', self.cost)

```


3. 模型优化

```
# 建立模型优化过程，训练时使用 train_op
with tf.name_scope('optimize'):
    self.lr = tf.placeholder(tf.float32, [])
    tf.scalar_summary('learning_rate', self.lr)

    optimizer = tf.train.AdamOptimizer(self.lr)
    tvars = tf.trainable_variables()
    grads = tf.gradients(self.cost, tvars)
    for g in grads:
        tf.histogram_summary(g.name, g)

    grads, _ = tf.clip_by_global_norm(grads, args.grad_clip)

    self.train_op = optimizer.apply_gradients(zip(grads, tvars))
    self.merged_op = tf.merge_all_summaries()
```

4. 模型迭代训练

模型训练函数（train）首先初始化参数，同时启动 Saver 将训练好的模型保存到文件（demo_seq2seq_model.ckpt）中，根据配置好的迭代次数进行训练。

```
def train(data, model, args):
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        saver = tf.train.Saver()
        writer = tf.train.SummaryWriter(args.log_dir, sess.graph)

        # 可视化 TensorBoard
        config = projector.ProjectorConfig()
        embed = config.embeddings.add()
        embed.tensor_name = 'rnnlm/embedding:0'
        embed.metadata_path = args.metadata
        projector.visualize_embeddings(writer, config)

        # 定义 max_iter 迭代次数
        max_iter = args.n_epoch * \
            (data.total_len // args.seq_length) // args.batch_size

        # 迭代训练
```

```

for i in range(max_iter):
    learning_rate = args.learning_rate * \
        (args.decay_rate ** (i // args.decay_steps))
    x_batch, y_batch = data.next_batch()
    feed_dict = {model.input_data: x_batch,
                  model.target_data: y_batch,
                  model.lr: learning_rate}
    train_loss, summary, _, _ = sess.run([model.cost, model.merged_op,
                                          model.last_state,
                                          model.train_op],
                                          feed_dict)

    # 调试信息，打印训练过程及损失率
    if i % 100 == 0:
        writer.add_summary(summary, global_step=i)
        print('step:{}/{}, loss rate:{:4f}'.format(i,
                                                    max_iter, train_loss))
    if i % 2000 == 0 or (i + 1) == max_iter:
        saver.save(sess,
                   os.path.join(args.log_dir,
                                'demo_seq2seq_model.ckpt'),
                   global_step=i)

```

9.3.5 生成 seq2seq 句子

根据训练好的模型生成测试结果，将生成句子的函数放置在 `generate` 函数中。

```

def generate(data, model, args):

    saver = tf.train.Saver()
    with tf.Session() as sess:
        ckpt = tf.train.latest_checkpoint(args.log_dir)
        print(ckpt)
        saver.restore(sess, ckpt)

    # 注意：如果训练集数据量很小，不包含 prime 的测试语句
    # 可能会因为 meta 中不存在相应字符的映射，导致生成语句失败
    prime = u'基于大数据的智能运维实践'
    state = sess.run(model.cell.zero_state(1, tf.float32))

```

```

for word in prime[:-1]:
    x = np.zeros((1, 1))
    x[0, 0] = data.char2id(word)
    feed = {model.input_data: x,
            model.initial_state: state}
    state = sess.run(model.last_state, feed)

word = prime[-1]
sentence = prime
# 生成句子
for i in range(args.gen_num):
    x = np.zeros([1, 1])
    x[0, 0] = data.char2id(word)
    feed_dict = {model.input_data: x, model.initial_state: state}
    probs, state = sess.run([model.probs, model.last_state], feed_dict)
    p = probs[0]
    word = data.id2char(np.argmax(p))
    print(word, end='')
    sys.stdout.flush()
    time.sleep(0.05)
    sentence += word

return sentence

```

在 `generate` 函数中, 首先通过 TensorFlow 的 `saver` 对象加载模型文件, 然后逐一生成句子。

9.3.6 运行演示

通过上面的模型训练过程, 可以定义主函数(`main`)。下面的代码演示了主函数的核心功能, 当 `flag` 被设置为 0 时进行模型训练, 设置为 1 时进行 `demo` 的演示 (生成句子)。

```

def main(flag):

    # 读取系统配置参数
    args = loadParam()

    # 加载数据集
    data = loadDataset('dataset/data.txt', args)

    # 加载模型
    model = Model(args, data, flag=flag)

    # flag 为 0 执行模型训练, flag 为 1 则执行 demo 演示

```

```
run_fn = generate if flag else train
# 运行
run_fn(data, model, args)
```

Python 的入口函数定义如下：

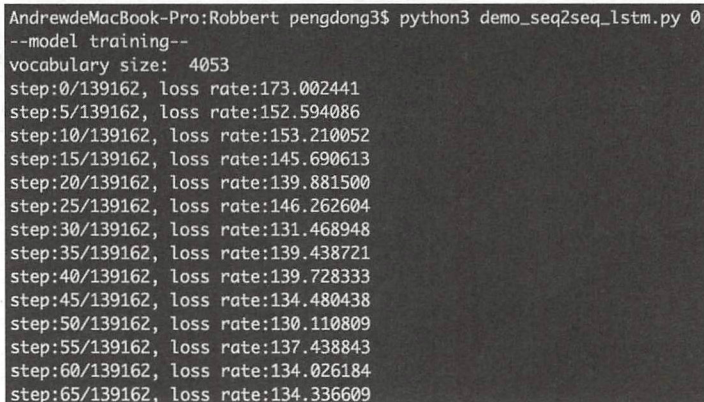
```
if __name__ == '__main__':
    msg = """
        model training please input:
        python3 demo_seq2seq_lstm.py 0
        generate please input:
        python3 demo_seq2seq_lstm.py 1
        """

    if len(sys.argv) == 2:
        flag = int(sys.argv[-1])
        print('-generate-' if flag else '-model training-')
        main(flag)
    else:
        print(msg)
        sys.exit(1)
```

执行模型训练，可以直接运行以下命令：

```
# 使用 Python 3.*, TensorFlow 0.12
# 演示代码保存在 demo_seq2seq_lstm.py 中
python3 demo_seq2seq_lstm.py 0
```

训练过程如图 9-7 所示。训练完成后，会在 logs 文件夹下产生模型文件，如图 9-8 所示。



```
AndrewMacBook-Pro:Robbert pengdong3$ python3 demo_seq2seq_lstm.py 0
--model training--
vocabulary size: 4053
step:0/139162, loss rate:173.002441
step:5/139162, loss rate:152.594086
step:10/139162, loss rate:153.210052
step:15/139162, loss rate:145.690613
step:20/139162, loss rate:139.881500
step:25/139162, loss rate:146.262604
step:30/139162, loss rate:131.468948
step:35/139162, loss rate:139.438721
step:40/139162, loss rate:139.728333
step:45/139162, loss rate:134.480438
step:50/139162, loss rate:130.110809
step:55/139162, loss rate:137.438843
step:60/139162, loss rate:134.026184
step:65/139162, loss rate:134.336609
```

图9-7 执行seq2seq的模型训练过程


```

287 4 6 16:00 checkpoint
9163196 4 6 15:40 demo_seq2seq_model.ckpt-0.data-00000-of-00001
1327 4 6 15:40 demo_seq2seq_model.ckpt-0.index
317660 4 6 15:40 demo_seq2seq_model.ckpt-0.meta
9163196 4 6 15:36 demo_seq2seq_model.ckpt-1002.data-00000-of-00001
1327 4 6 15:36 demo_seq2seq_model.ckpt-1002.index
317660 4 6 15:36 demo_seq2seq_model.ckpt-1002.meta
9163196 4 6 15:47 demo_seq2seq_model.ckpt-2000.data-00000-of-00001
1327 4 6 15:47 demo_seq2seq_model.ckpt-2000.index
317660 4 6 15:47 demo_seq2seq_model.ckpt-2000.meta
9163196 4 6 15:53 demo_seq2seq_model.ckpt-4000.data-00000-of-00001
1327 4 6 15:53 demo_seq2seq_model.ckpt-4000.index
317660 4 6 15:53 demo_seq2seq_model.ckpt-4000.meta
9163196 4 6 16:00 demo_seq2seq_model.ckpt-6000.data-00000-of-00001
1327 4 6 16:00 demo_seq2seq_model.ckpt-6000.index
317660 4 6 16:00 demo_seq2seq_model.ckpt-6000.meta

```

图9-8 训练得到的模型文件集日志

TensorFlow 提供了可视化工具 TensorBoard，通过 TensorBoard 可以非常方便、简捷地进行可视化查看，甚至修改参数。执行以下命令可以运行 TensorBoard 服务：

```
# 使用 Python 3.*, TensorFlow 0.12, 打开 TensorBoard 可视化工具
tensorboard --logdir=logs
```

TensorBoard 默认会启动 Web Server 并打开本机 6006 端口，通过访问 <http://localhost:6006> 使用可视化工具，如图 9-9 所示。

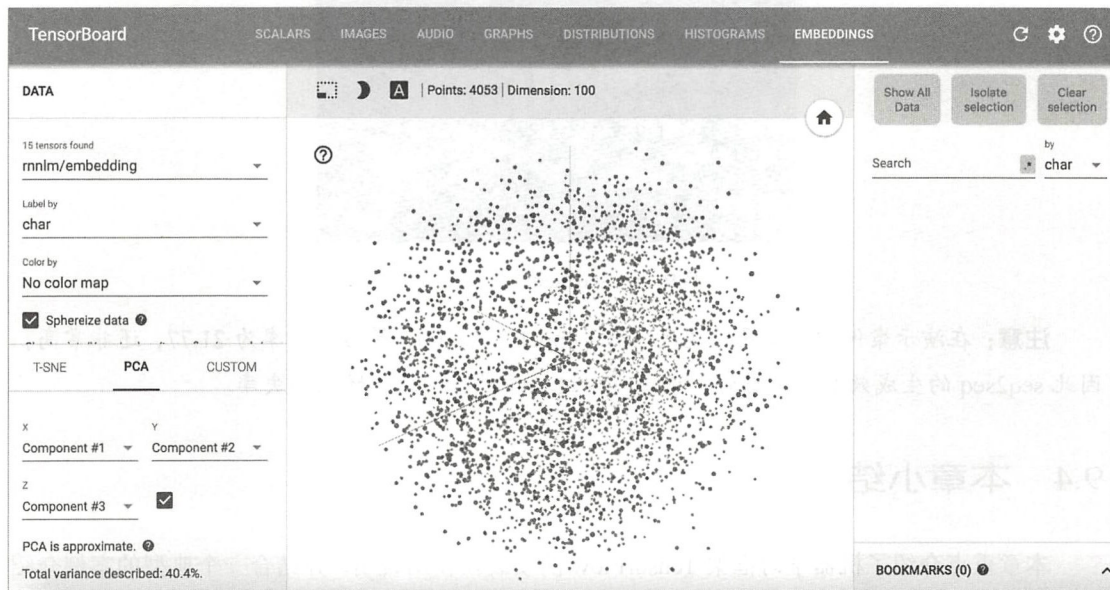


图9-9 TensorBoard可视化工具

TensorBoard 也支持展示模型 Graph 拓扑，图 9-10 展示了模型训练的图。

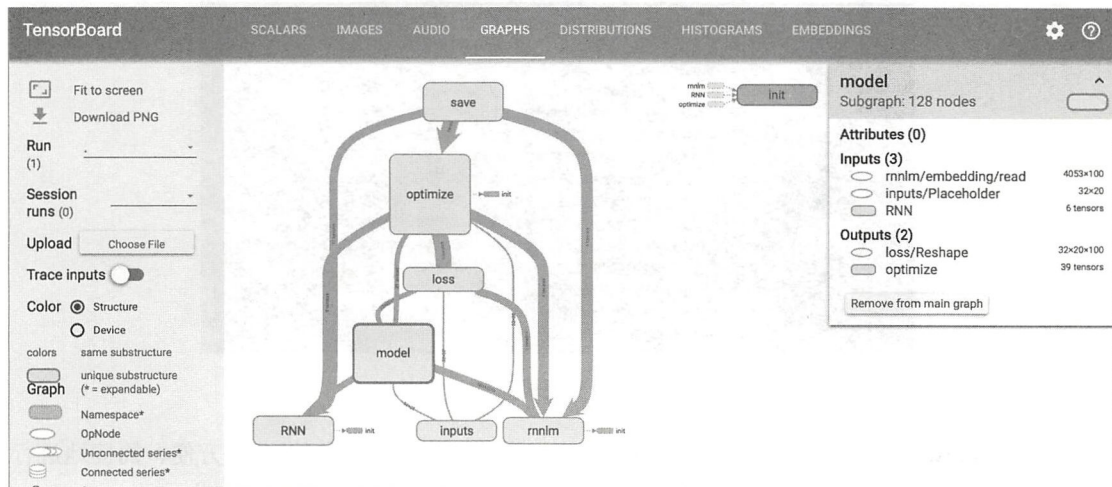


图9-10 TensorBoard的GRAPHS页面

最后，可以查看执行结果。如图 9-11 所示，通过训练好的模型，生成了 100 个字的短句。

```
./logs/demo_seq2seq_model.ckpt-10036
```

却亲操城的鸽落
我牵着你的爱情
我们溜了多张
我却在一双
白什么去况子 依然马了望握上
故事在招惹 秋晨花心一枚飘外
你们畏此 给醒在左人没有我
想是一场悲剧
我只当心心步过改变
就是你讲
我轻轻的

图9-11 seq2seq的生成结果

注意：在演示案例中，设置的迭代次数（epoch）为 100，最后损失率为 21.77，还非常高，因此 seq2seq 的生成效果还不够好，读者可以进一步进行优化，降低损失率。

9.4 本章小结

本章重点介绍了机器学习框架 TensorFlow 的安装及使用说明，并结合一个典型的案例介绍了如何在 TensorFlow 中训练和使用 seq2seq 模型。作为 Google 开源的重量级机器学习框架，TensorFlow 在分布式模型训练中有丰富的技术支撑，开发者可以很方便地在 TensorFlow 中集成自己的算法，极大地减少了工程量。

当然，机器学习框架非常多，各有所长，读者可以根据自己的习惯进行选择。

9.5 参考文献

[1] TensorFlow 官方网站: <https://tensorflow.org>

[2] Cho et al.. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014

[3] Vinyals et. al.. A Neural Conversational Model[J]. Computer Science, 2015

第3篇

运维新时代：智能运维技术详解

AIOps 正在以一种全新的姿态变革和影响着传统运维体系，也给运维技术方向注入了新的活力和想象空间。如果说运维平台解决了自动化的问题，那么人工智能则在平台的基础上解决了智能化的问题，解决了如何让机器进行判断和决策的问题。我们知道，“人工智能(AI, Artificial Intelligence)”这一术语自麦卡赛、明斯基、罗切斯特等知名科学家于 1956 年首次提出以来，已经经历了半个多世纪的发展，在机器视觉、指纹识别、人脸识别、视网膜识别、虹膜识别、掌纹识别、专家系统、自动规划、智能搜索、定理证明、博弈、自动程序设计、智能控制、机器人学、语言和图像理解、遗传编程等众多领域都有实际的应用并发挥着价值。越来越多的企业开始关注如何使用 AI 来解决生产环节的效率问题，以及如何提升对客户理解。

随着硬件技术的不断升级，运算能力也从传统的以 CPU 为主导发展到以 GPU 为主导，这使得 AI 发生了很大变革。算法技术的更新助力人工智能的兴起，早期的算法一般是传统的统计算法，如 20 世纪 80 年代的神经网络，90 年代的浅层，2000 年左右的 SBM、Boosting、Convex 的 Methods 等。随着数据量的增大，计算能力变得更加强大，深度学习的影响也越来越大。自 2011 年之后，深度学习的兴起，带动了人工智能发展的高潮。

在运维技术领域，新时代的要求是在大数据基础之上，如何高效、快速、准确地捕获系统异常，如何快速诊断，如何进行报警的准确预测和评估等。在这里，Ops 解决了资源维护、机器和服务的基本运维与保障问题；DevOps 解决的是如何构建工具和平台进行自动化，提升运维效率的问题；SRE 解决的是如何保障系统具有高可用性和稳定性的问题；AIOps 解决的是如何在大数据场景下对实时数据进行处理和分析，以及智能化决策的问题。

本篇主要讨论智能运维的常见方法和模型，以及在具体应用中的使用场景。本篇主要分为以下几个章节：

- 第 10 章 数据聚合与关联技术
- 第 11 章 数据异常点检测技术
- 第 12 章 故障诊断和分析策略
- 第 13 章 趋势预测算法

第 10 章

数据聚合与关联技术

在智能运维技术领域，对数据的分析处理非常关键，通常多维数据需要降低维度，以降低计算和存储成本。对时序数据的处理大致分为两类。

1. 单系列时序数据聚合

单系列是指同一个类别组成的系列，比如订单表中订单时间这个维度。聚合的含义就是汇总，对一组数据通过一系列运算得出单一值的过程，这里的运算是指在固定周期内（10s、60s 等）对实时、时序数据流的运算。还有一类是基于数据库、数据仓库或者其他已存储的数据（CSV 格式文件、分布式文件系统 HDFS 等）的离线分析，比如利用 Pandas、Dask 分析数据，利用 Spark、Hive 处理大数据等，这些都会涉及聚合数据。

2. 多系列时序数据关联

多系列是指多个类别组成的系列。数据关联的结果是将不同系列的数据整合到一起，方便后续的数据应用。

本章主要针对时序数据介绍相关的数据聚合与关联技术。

10.1 数据聚合

本节关注的是对实时、时序数据（监控数据和运维事件）的聚合。而数据分析属于离线计算模式，是对数据的深加工、二次利用，这里不做太多的叙述。

聚合数据有两个层面，一是对数据的聚合运算（计数、平均、抽样等）；二是多维度的聚合（在实际应用中维度通常指时间、地点、业务线、服务、接口等，它们也可以被称作“标签”，用来标注看待数据的不同角度）。

10.1.1 聚合运算

常见的进行时序数据聚合运算的工具如下。

- Statsd (<https://github.com/etsy/statsd>)
- Statsite (Statsd 的 C 语言实现, <https://github.com/statsite/statsite>)
- Collectd (<https://github.com/collectd/collectd>)

支持的数据/事件类型如下。

- Key/Value: 不进行聚合运算, 直接存储原值。
- Sampling: 抽样数据。
- Gauges: 固定值。
- Counter: 计数类数据, 一般会进行求和、累加。
- Timers: 时间型数据。
- Sets: 集合型数据。
- Multi-Metric Packets: 批量数据。

针对 Timers 类型的数据, 可以做如下运算:

- Mean: 求平均值。
- Min/Max: 求最小值/最大值。
- Standard deviation: 求方差。
- Median,P95,P99: 求中位数、95%值、99%值。
- Histograms: 求直方图/柱状图。
- Sum: 求累计值。

数据聚合在整个运维监控系统中所处的位置如图 10-1 所示。

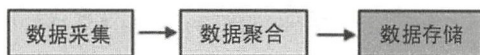


图10-1 数据聚合在整个运维监控系统中所处的位置

可以说，如果没有聚合这一层，我们就不得不存储所有的原始数据（在某些场景下需要原始数据，但在运维监控工作中基本不需要），读取时也需要读出原始数据后再计算结果，这样写入量和读取量就会比不做聚合时大得多，在海量数据场景下，这对系统复杂度和稳定性都是一个考验。

Statsd/Statsite 工作模式相对简单，启动后监听 UDP/TCP 某个端口，客户端将 UDP/TCP 数据包直接发送到对应端口即可。无论是使用读取日志的工具（比如 Logtailer）来发送数据，还是在应用程序中通过 UDP/TCP 的方式发送数据包都非常方便。发送数据包的内容格式如下：

```
<metricname>:<value>|<type>
```

如下是最简单的指标发送命令格式，发送一条计数类指标到 Statsd：

```
echo "foo:1|c" | nc -u -w0 xx.xx.xx.xx 8125
```

说明：-u 参数表示 UDP 协议。

这样就完成了一个指标的发送，Statsd/Statsite 进程收到数据包后，会按照预先设定的计算周期聚合这些数据，计算出这些数据的总和、最大值、最小值、均值、中位数、P95 值和 P99 值等，然后发送到后端。Statsd/Statsite 支持的后端比较丰富，包括：

- Graphite
- InfluxDB
- Ganglia/Zabbix
- Nagios
- Librato
- CloudWatch
- OpenTSDB
- Elasticsearch
- Riemann/Bosun

○ Console

后端存储这些指标/数据后,可以转化成图表(典型的如 Graphite/Grafana),如图 10-2 所示。

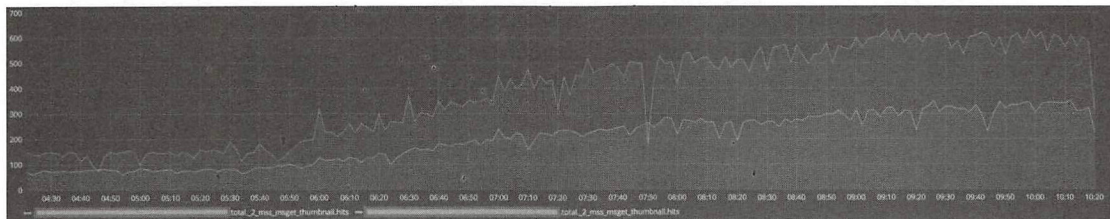


图10-2 Grafana展示效果

另外,也可以转化为告警事件(Nagios),存储在时序数据库中进行其他分析(如 OpenTSDB/InfluxDB),以及进入复杂事件处理流程(如 Riemann/Bosun)进行关联分析等。

Statsd 作为时序数据的聚合工具具有以下优势。

- 简单: 非常容易获取的应用程序, Statsd 协议是基于文本的,可以直接写入和读取。
- 低耦合性: 基于后台程序运行的应用程序,采用 UDP 协议,收集指标和应用程序本身之间没有依赖。
- 占用空间小: Statsd 客户端非常轻便,不带任何状态,甚至不需要客户端,只要按照协议发送文本指标到 Statsd 指定端口即可。
- 支持多种语言: 有基于 Ruby、Python、Java、Erlang、Node、Scala、Go、Haskell 等语言的客户端。

Collectd 是另一种风格,相对于 Statsd 的独立(相对独立,在大多数情况下 Statsd 都是作为服务单独部署的,当然也有把 Statsd 部署在客户端的情况),通常 Collectd 被当作 Agent 部署在需要采集信息的客户端,再加上它有特别丰富的插件,使得在 Agent 端进行数据聚合也成为一种选择(小米监控采取的就是这种策略,Open-Falcon 的 Agent 完成了很多聚合工作,这样在大数据量下后端的压力较小)。

但从另外一个角度来讲,任何 Agent 对系统都或多或少有侵入性,在有些对安全性要求较高的场合,采用 Agent 方式可能很难部署。

Collectd 和 Statsd 一样,只是数据聚合工具,没有绘图、告警、分析数据等功能,所以它也依赖第三方工具来完成这些工作。如图 10-3 所示是 Collectd 的 Plugin 列表(<https://collectd.org/>)

wiki/index.php/Table_of_Plugins), 其中标记为 Write 的就是它支持的后端。

Write Graphite plugin	Write	collectd.conf(5)	5.1
Write HTTP plugin	Write	collectd.conf(5)	4.8
Write Kafka plugin	Write	collectd.conf(5)	5.5
Write Log plugin	Write	collectd.conf(5)	5.5
Write MongoDB plugin	Write	collectd.conf(5)	5.1
Write Redis plugin	Write		5.0
Write Riemann plugin	Write	collectd.conf(5)	5.3
Write Sensu plugin	Write	collectd.conf(5)	5.5
Write TSDB plugin	Write	collectd.conf(5)	5.5

图10-3 Collectd支持的后端

10.1.2 多维度聚合

关于多维数据前面章节介绍过,就是以什么角度来看待、检索、处理数据,典型的多维数据模型如图 10-4 所示。

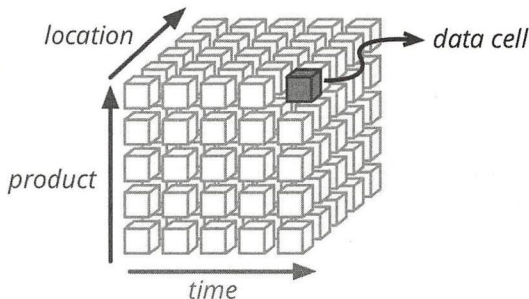


图10-4 典型的多维数据模型^[2]

这里只展示了数据的三个维度:时间、地点、产品,但是在实际应用中,尤其是在复杂业务模型中,则有很多维度。

我们来看看监控和运维活动中的多维数据,以 Statsd 的 Metric 协议为例:

```
Nginx_qps:100|c
```

这看上去是一维数据,指标是 QPS,维度就是 Nginx。

```
Openapi.profile.uncore.tc.db.xxxx.total_count:230|c
```

这看上去就包含了多个维度。

- Openapi: 业务线。
- profile: 性能类指标。
- uncore: 非核心业务。
- tc: 机房名。
- db: 数据库服务。
- xxxx: 数据库域名。
- total_count: 指标名。

用“.”分隔的 Metric 组织方式, 将所有的维度信息放在了指标名中, 这样做的好处是很直观, Statsd 可以按照“.”来分类聚合。使用 Graphite/Grafana 可以很方便处理这样的 Metric 组织方式。以上面的 Metric 为例, 可以通过下面的方式在 Graphite/Grafana 中组织一个 Metirc:

```
Openapi.profile.*.{tc,yf,xd}.db.{m7365*,m3360*}.total_count
```

图 10-5 展示了一个维度聚合示例。

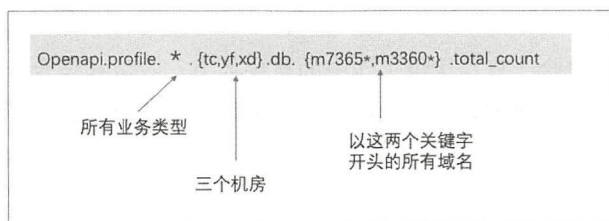


图10-5 维度聚合示例

可以看到, Graphite/Grafana 可以自由使用通配符、集合在各个层级进行不同维度的聚合。如果使用 Statsd+Graphite/Grafana 进行监控, 这种指标的设计方式可以将写入和读取无缝联合起来, 中间不用再进行数据转换。

多维数据的另一种组织方式是用 Tag 来扩展维度, 比如:

```
Metric: total_count
```

```
IP: xx.xx.xx.xx
```

```
Tags:{interface:user_show,service_pool:webv2_action,location:yf}
```

```
Value:2000
```

Type: count

在 Tags 字段中，标注了 interface（接口）、service_pool（服务池）、location（机房）三个维度。相对于在指标名里包含维度信息，Tags 方式不受指标名长度的限制（指标名超过 255 字节会导致在某些 KV 存储系统中无法使用，比如 Memcached），可以任意扩展维度。一些时序数据库（比如 OpenTSDB）就需要按照“Tags”来存储指标，如果使用 OpenTSDB，那么对于上面介绍的在指标名中包含维度信息的情况，就需要增加一层（在 Statsd 层或 Relay 层）来专门拆分指标、提取 Tags，然后转换成 TSDB 格式存储。

带 Tags 类的指标信息处理就不太适用 Statsd 直接计算了，一些实时流式计算框架如 Storm 等可以用来处理这类较复杂的 Metric 的聚合和运算，这就需要一定的开发量。

10.2 降低维度

上面刚刚说了可以扩展多维数据，然而，维度多了也会带来麻烦。比如告警这个场景，当一个业务/服务有问题时，可能会触发多个维度的事件，如接口维度、机房维度、服务池维度等，如果都触发了告警，就变成了噪音，你没有办法快速定位根本原因，一天收到成千上万个告警，基本上就等于告警无效了。

告警事件的收敛和聚合一直是运维工作中的重要环节。收敛和聚合一方面可以减少噪音和干扰；另一方面可以确定主要因素、定位故障。如果每天有上万个报警事件，那么如何做才能收敛到人类可处理的程度（少于 10 个）呢？接下来我们将介绍一些常用的方法。

- 将告警聚合成关联“事件”。
- 合并重复的告警。
- 自动恢复策略。
- 告警分类。

重复告警的合并就是指当故障发生时，同一个监测点在每个监测周期内都会发现异常并触发告警，需要按照一定的策略（一般是时间窗口策略，比如在 5 分钟内最多发送多少条告警信息，每天上限是多少等）进行告警合并；自动恢复策略就是将告警和相应的干预手段联合起来，由告警触发一些干预动作（如自动扩容、重启、流量切换等），这样可以直接消除告警，减少人工干预。这两种方法的原理都相对简单。

10.2.1 将告警聚合成关联“事件”

在运维工作中告警因为维度多，绝对数量可能很大，但关联到的“事件”却是有限的。以微博为例：上行功能有发、转、评、赞，下行功能有 Feed，就这 5 大功能，其他数千个功能、组件、资源都是为之服务的，任何维度的告警都可以被归类到这 5 大功能（具体影响到哪个功能）上。当然，这是理想的情况，在实际工作中，除了这 5 大功能，开发和运维人员关心的维度还要稍微多一些。比如哪个机房的哪个服务池的哪个功能有问题，对这 5 大功能再增加几个维度，总体关注事件的数量仍然是一个有限的集合，这对运维人员来说就是可控的。

图 10-6 展示了一个按照功能聚合后的事件告警页面。

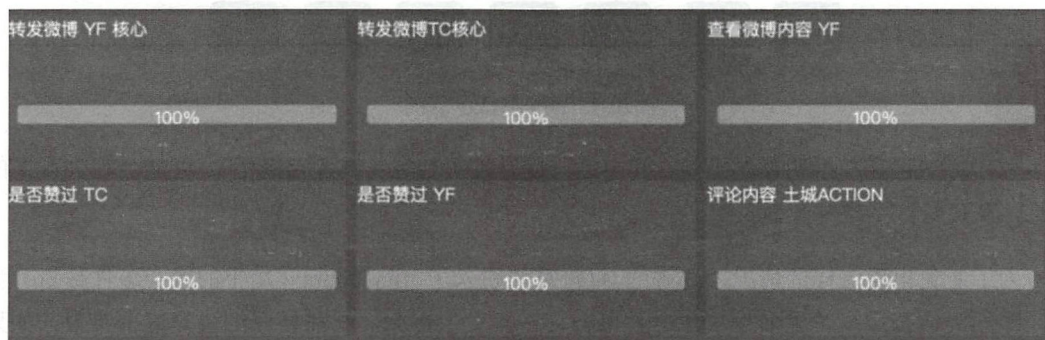


图10-6 按照功能聚合后的事件告警页面（对一些内部信息做了处理）

虽然这只是一部分，但比起各种告警的轰炸，已经是可以接受的程度了。“事件”是对业务、服务的抽象、聚焦，可以让我们重点关注业务本身，而不是被各种告警、噪音分散注意力。

大量、多维度的告警聚合为少量的“事件”就是降低维度，使之可以被正确处理。降低维度的主要方法就是聚类（Clustering），聚类算法有不少，比如 K 均值聚类、层次聚类、PCA 等。现在我们介绍的是概念聚类算法 AOI（Attribute Oriented Induction，基于属性的归纳），如图 10-7 所示。

AOI 算法的本质就是不断地抽象化（将某个属性的值替换为更抽象的值），抽象的层次由开发和运维人员定义。如图 10-7 所示，当某个 IP 写入队列接口耗时变长（超过阈值）时，这个告警的 IP 属性会被更抽象的“服务池”替换，同样，服务池也可以被“机房”替换，最终各种维度的告警经过几层抽象会被聚合、收敛为少量的事件。

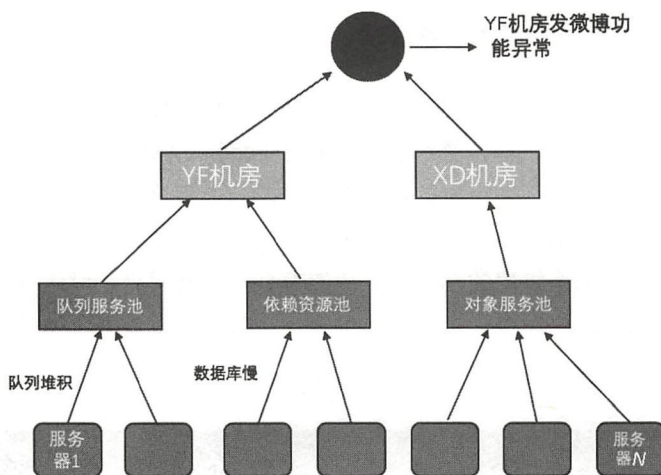


图10-7 AOI算法示意图

AOI 算法也存在很明显的不足，其中之一便是过于抽象（或者叫概化），可能会损失太多的信息，从而导致结果毫无意义^[2]。

为了避免出现这样的情况，改进的 AOI 算法引入了 `min_count`（最小的聚类事件数量）参数，即同类告警最少发生多少次才会被聚合，一旦发生告警，就需要仔细选择这个参数：如果参数值太大，则会聚合掉一些根本原因而导致结果太过抽象和概化（比如微博有问题，这个结论对排查问题毫无帮助）；如果参数值太小，则一个原始告警可以匹配多个事件。

除了 `min_count`，可以再引入一个参数：`max_gen`（最大概化次数），标明一个告警最多只允许经过 `max_gen` 次概化，如果经过 `max_gen` 次概化，仍然不能满足数量大于 `min_count` 的条件，则说明这些告警之间的相似性不高或者相似性高的告警不多，对这些数据将不作处理，留原形式提交给管理员，供管理员自行处理。

从结果来看，经过 AOI 算法聚合后，应该已经消除了大部分噪音，让运维人员只关注重要的事件，不被各类噪音所干扰。

10.2.2 减少误报：告警分类

上面介绍的聚类算法，可以有效地减少告警数量，但对于由网络抖动、流量峰值等引发的不需要人工干预的、自愈性事件，效果就不太好了。过多过于频繁的误报，会使运维值班人员精神上产生疲惫，以至于最终对真正的告警不再敏感（警报疲劳）。

如果通过机器学习技术对一组已知的告警数据进行训练学习,生成分类模型(会不断调整、优化),然后模型对新接收的告警数据进行分类,是否会减少误报的概率呢?

下面介绍一种基于告警分类器的减少误报的方法。“分类”是人工智能和机器学习领域的重要算法,分类方法是一种对离散型随机变量建模或预测的监督学习算法,有监督的学习意味着需要有一些已标注的数据作为训练样本。通常会提供一个告警结果页面,供开发和运维人员来选择——处理、忽略、误报等选项可以将告警数据标签化,然后建立训练样本。分类器既可以由人工构建,也可以由机器学习技术自动构建。

分类算法有 K 近邻(KNN)、贝叶斯分类器(NB)、Logistic 回归(LR)、支持向量机(SVM)和随机森林(RF)等,有时候我们可能需要不同的算法交叉验证学习效果,如图 10-8 所示。

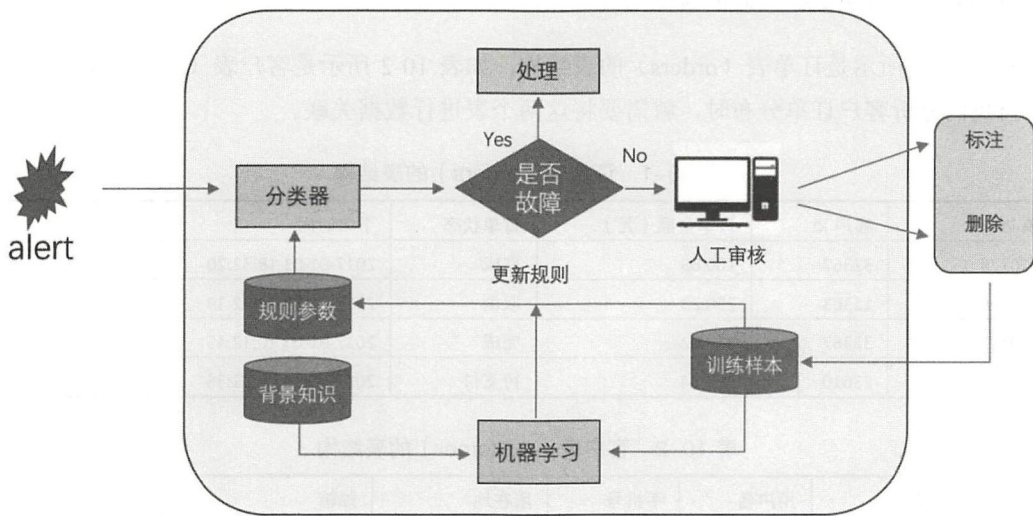


图10-8 自适应告警分类器的实现

关于告警收敛的整体流程如图 10-9 所示。

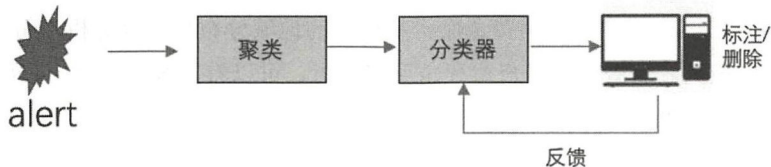


图10-9 告警收敛的整体流程图

10.3 数据关联

在大数据时代，金融、互联网、自动化等方向的数据量爆炸性激增，同时数据的维度也相应地变得复杂，各行各业都在从数据中寻找业务的增长点、用户的画像以及更智能的推荐。为了实现上述业务或者需求，需要通过关联很多数据维度来计算。对于当代的计算机来说，在数据量小的情况下很容易快速实现数据关联。比如数据分析师通过 Excel 就可以进行数据关联，还可以通过写简单的程序在笔记本电脑上进行数据关联处理，但是当数据量达到百亿级别时，则无法把数据完全加载到内存中进行计算。

数据关联一般指根据一定的规则和关联关系连接各类数据以便形成新的数据，MySQL 数据库中的 join 操作就是数据关联的典型用例。

如表 10-1 所示是订单表（orders）的表结构，如表 10-2 所示是客户表（customer）的表结构，当我们分析客户订单分布时，就需要将这两个表进行数据关联。

表 10-1 订单表（orders）的表结构

订单 id	客户 id	订单金额（元）	订单状态	下单时间	备注
20170125	32367	100.68	完成	2017-01-01 18:32:20	
20170126	11363	230.23	取消	2017-01-02 08:02:19	
20170127	32367	112.02	完成	2017-01-03 10:12:45	
20170128	23610	802.74	待支付	2017-01-04 22:32:36	

表 10-2 客户表（customer）的表结构

客户 id	昵称	用户名	手机号	所在地	邮箱	备注
32362	张三	zhangsan	135xxx	北京	zhangsan@xxx	
32365	李四	lisi	152xxx	深圳	lisi@xxx	
32367	王五	wangwu	180xxx	上海	wangwu@xxx	

经过关联后可以得到“王五”这个用户完成的订单数量为 2 个（订单 id 为 20170125 和 20170127），共贡献了 212.70 元的消费金额。从用户的地域分布角度来分析，可以得出上海用户量最大，贡献度最高。从潜在的客户留存角度来分析，可以看出客户 id 为 11363 的客户有消费意愿，需要进一步挖掘。

在实际中数据关联会非常复杂，可能涉及多种数据源，甚至是异构数据的关联。上面的案例属于离线数据关联，即需要关联的多个数据都已经存在。而在监控系统中，经常遇到的情况

是实时数据关联，下一节我们将结合微博广告的具体案例来探讨如何进行实时数据关联。

10.4 实时数据关联案例

对实时性要求高的如广告 CTR 预估、运维监控、广告投放效果评估等产品需求来说，Hadoop 离线数据处理已经无法满足要求。

例如广告互动、负反馈等数据总体晚于曝光数据到达，由于实时数据到达的时间点不确定，理论上实时数据很难做到百分之百的关联。目前实时数据关联存在以下问题。

- 关联数据到达时间不一致。数据到达时间不一致，导致其中一部分先到的数据需要等待，数据等待的时间长短、实时计算的时间窗口大小都会直接影响到数据关联的准确性。
- 关联数据严重倾斜。需要进行数据关联的数据量相差甚远，例如微博某广告产品曝光数据在亿级别，而对应的广告互动的数据可能在千万级别，这对于有效、快速地进行数据关联，减少不必要的数据检索，是一个巨大的挑战。
- 实时计算引擎不适合缓存大量的数据集。实时计算引擎基本都是采用内存计算的，为了提高实时计算的性能和效率，在需要关联的数据没有到达之前，部分数据需要在内存中等待。在进入实时计算引擎的数据 QPS 较大的情况下，很容易造成计算引擎内存溢出。

10.4.1 设计方案

对于关联数据到达时间不一致的情况，要提高数据关联的准确性，就需要先到的数据等待。然而，在一定时间范围内进入实时计算引擎的数据量通常相对较大，比如 1 分钟可能有上千万条数据，为了进行数据关联，就需要将这部分数据进行缓存。由于大量的数据被缓存到内存，可能会导致实时计算引擎内存溢出，同时当应用程序意外终止后，可能导致因内存数据丢失而带来数据缺失问题。对于这种情况，通常的解决方案是借助外存。

数据关联整体架构示意图如图 10-10 所示。

从图 10-10 可以看出，曝光数据和互动数据按 `mark_id`（唯一标识广告，同时含有广告出价等信息的签名串）进行关联。由于曝光数据早于互动数据到达，因此先将曝光数据写入 HBase，当互动数据到达时再到 HBase 中查询，并写入新的互动数据，得到的最终数据则关联上了曝光

和互动。这里使用 Kafka 作为数据中转的消息队列。

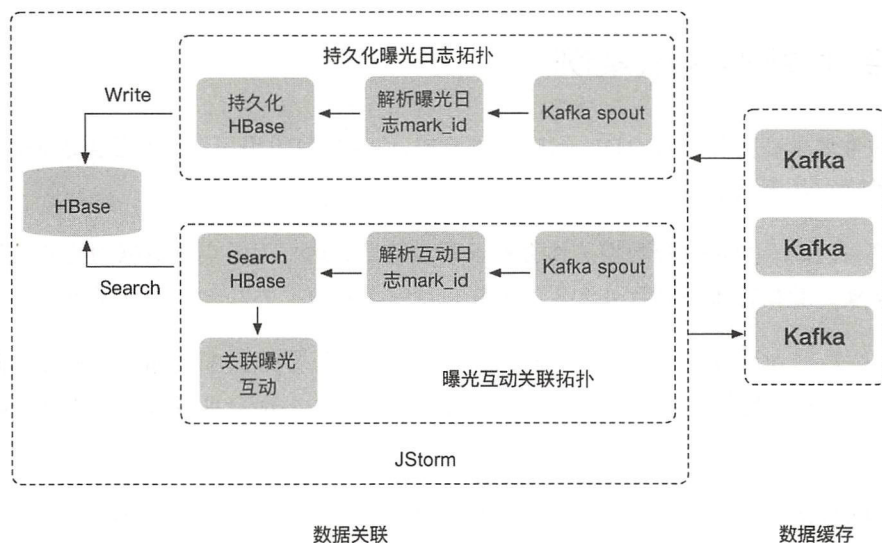


图10-10 数据关联整体架构示意图

1. 外部存储选型（Redis、HBase）

进行数据关联的外部存储首选 Redis。Redis 查询耗时在毫秒级，将数据存储在内存中，支持数据落地，可以做到数据保障。Redis 经常被用于缓存数据，其在性能上满足大规模数据的查询需求。

其次可以选择 HBase。HBase 分布式存储的写吞吐量高，查询响应耗时在 50 毫秒左右。HBase 数据存储在 Hadoop 分布式文件系统上，支持海量的数据存储，数据存储在磁盘上。

外部存储选型是基于公司现有的存储所做的选择，除了 Redis、HBase，还有 MongoDB、ClickHouse 等可以选择。经过以上分析可知，Redis 虽然查询性能较好，但它是基于内存存储的，相对于磁盘来说，消耗的资源较为昂贵，如果把上十亿、百亿级别的数据存储到 Redis 上，对于公司来说则成本较高；而 HBase 虽然拥有海量的存储，写吞吐量高，但其查询性能不能满足实时集群的要求，至少无法满足实时计算引擎每次的查询要求。通过分析 Redis、HBase 的优缺点，最终我们选择了 HBase 作为实时数据关联的存储和查询引擎。

在实际过程中，我们会综合使用缓存和外部存储方案，在计算节点上使用内存进行一定程度的数据缓存，以减小对 HBase 的查询压力。比如，在进行广告曝光数据和互动数据关联时，可以缓存 1 分钟的互动数据，然后再查询存储在 HBase 上的曝光数据，而不是对每一条互动数

据都进行 HBase 的查询关联。

2. 如何缓存数据

如上所述，我们需要在实时计算引擎节点缓存一部分数据，降低对 HBase 的查询次数。缓存数据是为了更好地进行本地数据查询，因此可以选择 HashMap 作为数据结构。需要注意的是，内存中的数据可能会因任务失败而丢失，因此应用程序应该设置一定的规则，防止存入 HashMap 的数据过大。

3. 如何提高数据关联的准确性

实时数据要关联的数据到达时刻不一致，为了提高数据关联的准确性，就需要对没有关联的数据进行多次关联。所以，可以将没有关联到的数据放入队列中通过单独的线程进行关联，以提高数据关联的准确性。

10.4.2 效果

从整体架构设计来看，我们使用 HBase 代替 Redis，使用磁盘存储替换纯内存存储，在满足需求的情况下降低了成本。引入缓存先进先出的队列，降低了 HBase 的读写压力。最终曝光和互动的实时数据关联准确率在 98% 以上（通过设置缓存时间窗口，可以在系统可用性和关联准确率上进行权衡）。

10.5 本章小结

本章重点介绍了数据聚合和数据关联技术，在数据聚合方面，讨论了聚合方法、多维数据聚合技巧，以及如何降低维度；在数据关联技术方面，则重点以微博广告的具体业务为例，讨论了如何在实时流场景下进行实时数据的关联。

10.6 参考文献

[1] <https://pythonhosted.org/cubes/introduction.html>

[2] Tadeusz Pietraszek, Axel Tanner. Data Mining and Machine Learning - Towards Reducing False Positives in Intrusion Detection

[3] 邓志龙. 浅谈对概念聚类 AOI 算法分析的改进. 陕西青年职业学院学报, 2010(3): 29-33

第 11 章

数据异常点检测技术

虽然计算硬件和软件的快速发展已经极大提高了应用程序的可靠性，但是在大型集群中仍然存在大量的软件错误和硬件故障。系统要求 7×24 小时不间断运行，因此，对这些系统进行持续监控至关重要。从数据分析的角度来看，这意味着不间断地监视大量的时间序列数据，以便检测潜在的故障或异常现象。由于实际中的系统异常或者软件 bug 可能会非常多，通过人工监控几乎是不可能的，因此非常有必要使用机器学习和数据挖掘技术进行自动化异常检测。

11.1 概述

异常值本质上是一个数据点。通常，大多数应用程序中的数据是由一个或多个反映系统功能的程序产生的。当底层应用程序以不正常的方式运行时，它会产生异常值。快速、高效地发现这些异常值非常有价值，异常值检测技术有非常广泛的应用，比如被应用在入侵检测、信用卡欺诈、传感器事件、医疗诊断、执法等领域。

通常所说的异常大致分为异常值、波动点和异常时间序列三类。

1. 异常值 (Outlier)

给定输入时间序列 x ，异常值是时间戳值对 (t, x_t) ，其中观测值 x_t 与该时间序列的期望值（即 $E(x_t)$ ）不同。

2. 波动点 (Change Point)

给定输入时间序列 x ，波动点是指在某个时间 t ，其状态（行为）在这个时间序列上表现出与 t 前后的值不同。

3. 异常时间序列 (Anomalous Time-series)

给定一组时间序列 $X = \{x_i\}$ ，异常时间序列 $x_i \in X$ 是在 X 上与大多数时间序列值不一致的部分。

在监控系统中,主要处理的是时序数据,而时序数据具有一定的特征,表 11-1 列举了时序数据的常见特征,表 11-2 列举了用于建模实验的常见指标。

表 11-1 时序数据的常见特征

特征	描述
周期(频率)	数据出现周而复始的现象
趋势	数据呈现上涨、下跌的走势
季节性	在一年或者更短的时期内在一个趋势线上重复性和可预测的变动
自相关	代表数据之间的相关依赖
非线性	时间序列中包含了非线性模型表示的复杂数据集
偏态	测量对称性,或更加明确地说,缺乏对称性
峰度	如果数据相对于正常分布达到峰值或平坦,则采取措施
林中小丘	衡量时间序列的长期记忆
李亚普诺夫指数	衡量附近轨迹的发散速度

表 11-2 用于建模实验的常见指标

指标	描述
Bias	误差的算术平均值
MAD	平均绝对偏差,也称为 MAE
MAPE	平均绝对百分比误差
MSE	误差的均方
SAE	绝对错误的总和
ME	平均误差
MASE	平均绝对比例误差
MPE	平均百分比误差

EGADS^[1]是 Yahoo 公司开发的一个灵活的、准确的、可扩展的异常检测综合系统。EGADS 框架与异常检测基准数据一起开源(请访问 GitHub 获取源码: <https://github.com/yahoo/egads>),帮助开发者开发新的异常检测模型。表 11-3 列举了常见的开源的异常检查模型(系统)。

表 11-3 常见的开源的异常检测模型(系统)

模型	描述
EGADS ExtremeLow—密度模型异常值	EGADS 基于密度的异常检测
EGADS CP	基于 EGADS 内核的波动点检测
EGADS KSigmaModel 异常值	EGADS 重新实现了经典的 k -西格玛模型
Twitter Outlier	基于广义的 ESD 方法的开源的 Twitter-R 异常检测库

续表

模型	描述
ExtremeI&II R 异常值	开源的异常值检测、阈值绝对值和残差检测异常
BreakOut Twitter CP	来自 Twitter 的一个库，基于 ESD 统计测试来检测变化点
ChangePt1 R CP	一个 R 库，实现各种主流的和专门的变化点方法，用于在数据中查找单个和多个变化点
ChangePt2&3 R CP	检测平均值和变量的变化

异常检测的目的是发现与大部分其他对象不同的对象^[2]。通常，异常点又被形象地称为离群点。在数据分布图中，它们远离其他数据点。从异常的成因来看，可以分为三类。

- (1) 数据来源于不同的类。例如，一群羊中混入了一匹马。
- (2) 自然变异。例如，一个正常身高的成年人在普通的群体中不是异常的，但是在篮球运动员群体中属于异常点。
- (3) 数据测量或收集误差。例如，由于实验机器故障，生成的实验结果跟机器正常运作时的结果大相径庭，产生异常数据。

另外，需要注意的一点是，异常基于一定的范围，分为全局异常和局部异常。例如，身高 2m 在全人类中是少见的、异常的，但是在职业篮球运动员中却是正常的，即是否异常跟比较对象的范围有关。接下来，我们就详细介绍异常检测方法。

11.2 异常检测方法

异常检测方法可以分归纳为三大类，每一类又包含了非常多的方法。

- 1. 基于统计模型的技术
- 基于统计模型的异常点检测技术将所有数据构建成一个数据模型，其认为异常点是那些与模型不能完美拟合的对象。
- 2. 基于邻近度的技术
- 通常可以在对象之间定义邻近性度量，并且许多异常检测方法都是基于邻近度的。异常对象是那些远离大部分其他对象的对象。
- 3. 基于密度的技术
- 对象的密度估计可以相对直接计算，特别是当对象之间存在邻近性度量时。当一个点的局

部密度显著低于它的大部分近邻时，可能会被看作是异常的。

11.2.1 基于统计模型的异常点检测

基于统计模型的异常点检测方法的基本思想是基于数据，构建一个概率分布模型，得出模型的概率密度函数。通常，异常点的概率是很低的^[2]。

1. 基于正态分布的一元异常点检测

假设数据集由一个正态分布产生，该分布用 $N(\mu, \sigma)$ 表示（如图 11-1 所示），其中， μ 和 σ 分别表示均值和标准差。数据落在 ± 3 个 σ 之外的区域的概率仅有 0.27%，落在 ± 4 个 σ 之外的区域的概率仅有 0.01%，几乎不可能发生，故将其判定为异常点。可以看出，异常点的判定与我们所选定的标准（几个 σ ）有关，不是固定的。

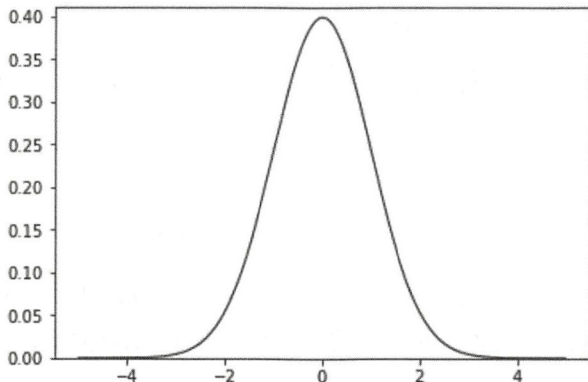


图11-1 $N(\mu, \sigma)$ 正态分布的概率密度图

2. 多元正态分布的异常点检测

对于多元高斯分布检测，我们希望使用类似于一元高斯分布的方法。例如，如果点关于估计的数据具有低概率，那么就把它分类为异常点。此外，我们希望能够用简单的检测方法如点到分布中心的距离来进行判定^[3]。

对于一个多维数据集 D ，假设 \bar{x} 是均值向量，那么对于数据集 D 中的其他对象 x ，从 x 到数据均值（质心）的 Mahalanobis 距离（马氏距离）为：

$$\text{Mahalanobis}(x, \bar{x}) = \sqrt{(x - \bar{x})^T S^{-1} (x - \bar{x})} \quad (11-1)$$

其中， x 为数据集 D 中的元素， \bar{x} 为数据均值， S 为协方差矩阵。容易证明：点到基础分布

的 Mahalanobis 距离与点的概率直接相关，等于点的概率密度的对数加上一个常数。因此，可以对 Mahalanobis 距离进行排序，距离大的，就可以认为是异常点。

接下来，我们通过一个具体案例（Python 实现代码见下文）来解释如何运用 Mahalanobis 距离来进行异常点检测。假设变量 y_1 和 y_2 均为一元正态分布随机变量， (y_1, y_2) 组成了二元正态分布数据集，称为 D 。现在，我们在数据集 D 中加入两个异常点： $A(7, 8)$ 和 $B(0, 10)$ ，形成新的数据集 S ，数据点分布如图 11-2 所示。

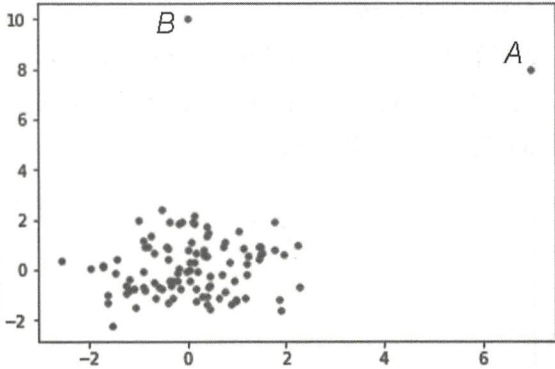


图11-2 二元正态分布数据集S的数据点分布

根据公式 (11-1)，计算数据集 S 中每个点到质心的 Mahalanobis 距离。计算结果如图 11-3 所示，正常点的距离均小于 3， A 点和 B 点的距离却大于 6，明显大于正常点的距离。因此，使用 Mahalanobis 距离法可以准确地检测出数据集 S 中的异常点 A 和 B 。

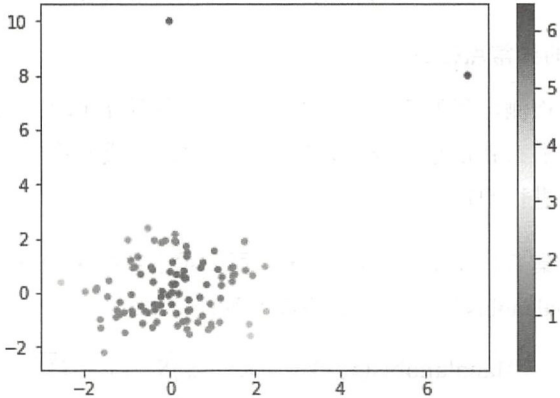


图11-3 数据集S中的点到质心的Mahalanobis距离

```
##马氏距离异常点检测实现代码
import numpy as np
import matplotlib.pyplot as plt
import math
##二元正态分布变量 y1,y2
mu = 0
sigma = 1
np.random.seed(0)
y1 = list(np.random.normal(mu, sigma, 100))
y2 = list(np.random.normal(mu, sigma, 100))
# 添加异常点 A (7,8) 和 B (0,10)
y1.append(7)
y1.append(0)
y2.append(8)
y2.append(10)
# Plot 图形
plt.scatter(y1, y2, color='blue', s =3)
plt.show()
# 计算马氏距离
X=np.array([y1,y2])
XT=X.T # 转置
XM = np.mean(XT, axis = 0) # 数据均值
# 求协方差矩阵及其逆矩阵
S=np.cov(X)
SI = np.linalg.inv(S)
# 根据马氏距离公式, 计算每个样本点到中心的距离, 得到结果集 dis
n=XT.shape[0]
dis=[]
for i in range(0,n):
    delta=XT[i]-XM
    d=np.sqrt(np.dot(np.dot(delta,SI),delta.T))
    dis.append(d)
# 将 dis[]与点的坐标关联起来
R = np.array([y1,y2,dis])
# 用马氏距离渲染
cm = plt.cm.get_cmap('RdYlBu')
sc = plt.scatter(R[0], R[1], c=R[2] , cmap=cm)
plt.colorbar(sc)
plt.show()
```


综上所述，两种基于统计模型的异常点检测方法，需要建立在标准的统计学技术（如分布参数的估计）之上。这类方法对于低维数据效果可能较好，但是对于高维数据，数据分布非常复杂，基于统计模型的检测效果会比较差。

11.2.2 基于邻近度的异常点检测

基于距离的异常点检测方法拓展了基本统计的思想，即如果一个对象是异常的，那么它远离大部分对象。这种方法比统计方法更一般、更普适。因为尽管数据集不满足任何统计分布模型，但是它仍能通过距离比较发现异常点。确定数据集的邻近性度量比确定它的统计分布要容易得多。

本节中，我们将为大家介绍最常用的 k -最近邻算法（ k -Nearest Neighbor），下文简称 KNN 算法。在 KNN 算法中，基本思想是一个对象的异常点得分由到它的 k -最近邻的距离给定。异常点得分的最低值是 0，最大值是距离函数的可能最大值，一般为正无穷。

异常点对 k 值的选取非常敏感，假设 $k=1$ ，则只取与对象点最近的点，求得距离。这样，如果数据集如图 11-4 所示，则会导致将圈中的两个异常点识别为正常的数据点，因为这两个点彼此的距离很近。但若 k 值选取太大，则点数小于 k 的集群中的点都有可能被识别为异常点，而错误地把一些小簇识别为异常点。

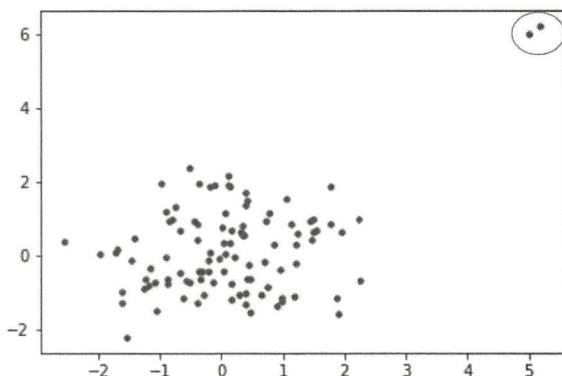


图11-4 k 值选取示意图

综上所述，KNN 算法是基于邻近度的算法，不需要对数据集进行统计模型的拟合，可以直接用距离来识别异常点。但是，这种基于距离的算法也有其明显的缺点：①时间复杂度为 $O(m^2)$ ，这意味着如果数据量比较大，会导致计算代价过高，效率低下；②对 k 值的选取非常敏感，并且 k 值的选取是全局的，不能处理具有不同密度区域的数据集。

11.2.3 基于密度的异常点检测

从基于密度的观点来说,异常点是低密度区域中的对象,即密度越低,异常点得分越高。那么如何定义密度,就直接决定了一个对象的异常点得分。定义密度的方法有以下三种。

(1) 逆距离:一个对象的密度为该对象周围 k 个最近邻的平均距离的倒数。这种逆距离定义密度的方法公式如下:

$$\text{density}(x, k) = \left(\frac{\sum_{y \in N(x, k)} \text{distance}(x, y)}{n} \right)^{-1} \quad (11-2)$$

其中, $N(x, k)$ 是 x 的 k 个最近邻的集合, n 是该集合的大小。

(2) 给定半径 d 内的个数:即一个对象周围的密度等于该对象指定半径 d 内对象的个数。这里需要谨慎选择 d 值,如果 d 值太小,那么大多数对象的异常点得分都会很低;如果 d 值太大,那么大多数异常点的密度与正常点类似,则有可能不被识别出来。

(3) 相对密度:即用点 x 的密度与它最近邻 y 的平均密度之比作为相对密度,数学公式如下:

$$\text{average relative density}(x, k) = \frac{\text{density}(x, k)}{\sum_{y \in N(x, k)} \text{density}(y, k) / n} \quad (11-3)$$

其中, $N(x, k)$ 是 x 的 k 个最近邻的集合, n 是该集合的大小。这种基于相对密度的异常点检测算法,可识别局部异常点。其简单的实现流程如下:

```
{指定 k 值, 作为最近邻个数}
for all x do
    确定 x 的 k 最近邻, 计作  $N(x, k)$ 
    使用  $N(x, k)$  中的对象, 确定 x 的密度  $\text{density}(x, k)$ 
end for
for all x do
    由公式 (11-3) 计算出  $\text{outlier score}(x, k) = \text{average relative density}(x, k)$ 
end for
```

综上所述,基于密度的异常点检测算法,与基于距离的检测算法是密切相关的,因为密度由距离来定义。所以,基于密度的算法时间复杂度也很高,也需要选择 k 值。但是,基于相对密度的算法可识别局部异常点。

11.3 独立森林

上一节介绍了几种传统的异常点检测方法，其基本思想是计算对象与周围对象的差异，无论是基于距离还是基于密度，都会得出一个异常点得分。这个分值越高，则代表该对象越有可能是异常点。本节中，我们将介绍独立森林（Isolation Forest）算法。

独立森林是南京大学周志华老师提出的一种异常检测算法，在工业界很实用，算法效果好，时间效率高，能有效处理高维数据和海量数据^[4]。

首先，要理解独立森林，就必须了解什么是独立树，下文简称 iTree。iTree 是一种随机二叉树，每个节点要么有两个子节点（称为左子树和右子树），要么没有子节点（称为叶子节点）。给定数据集 D ，这里 D 的所有属性都是连续型变量，iTree 的构成如下：

(1) 随机选择一个属性 A 。

(2) 随机选择该属性的一个值 $value$ 。

(3) 根据 A 对每条记录进行分类，把 A 小于 $value$ 的记录放在左子树上，把大于或等于 $value$ 的记录放在右子树上。

(4) 递归构造左子树和右子树，直到满足条件：①传入的数据集只有一条或多条一样的记录；②树的高度达到了高度阈值。

iTree 构造完成后，接下来对数据进行预测。预测的过程就是把测试记录从 iTree 根结点开始搜索，确定测试记录落在哪个叶子节点上。iTree 能检测异常的假设是：异常点一般都是非常稀有的，在 iTree 中很快会被分到叶子节点上。也就是说，在 iTree 中，异常值一般表现为叶子节点到根节点的路径 $h(x)$ 很短。因此，可以用 $h(x)$ 来判断一条记录是否属于异常值。

我们用一个归一化公式来计算异常指数 $S(x, n)$ ：

$$S(x, n) = 2^{\left(\frac{h(x)}{c(n)}\right)}$$

$$c(n) = 2H(n-1) - \left(\frac{2(n-1)}{n}\right)$$

$$H(k) = \ln(k) + \xi, \quad \xi \text{ 为欧拉常数} \quad (11-4)$$

其中， n 为样本的大小， $h(x)$ 为记录 x 在 iTree 上的高度。 $S(x, n)$ 的取值范围为 $[0, 1]$ ，越接近 1 表示是异常点的可能性越高，越接近 0 表示是正常点的可能性越高，如果大部分训练样本

的 $S(x, n)$ 都接近 0.5，则说明整个数据集没有明显的异常值。

当然，随机树是不稳定的，但是把多棵 iTree 结合起来，形成 iForest 就变得强大了。构建 iForest 的方法与构建随机森林的方法类似，都是随机采样一部分数据集来构造每一棵树，保证不同树之间的差异。但不同的是，我们需要限制采样样本的大小。这是因为：如图 11-5 所示，采样前正常值和异常值有重叠，采样后可以有效区分正常值和异常值。并且，还需要限制 iTree 的最大高度，因为异常值记录比较少，其路径长度也比较小。我们不需要将树的最大高度设置得很大，增加无意义的计算消耗。

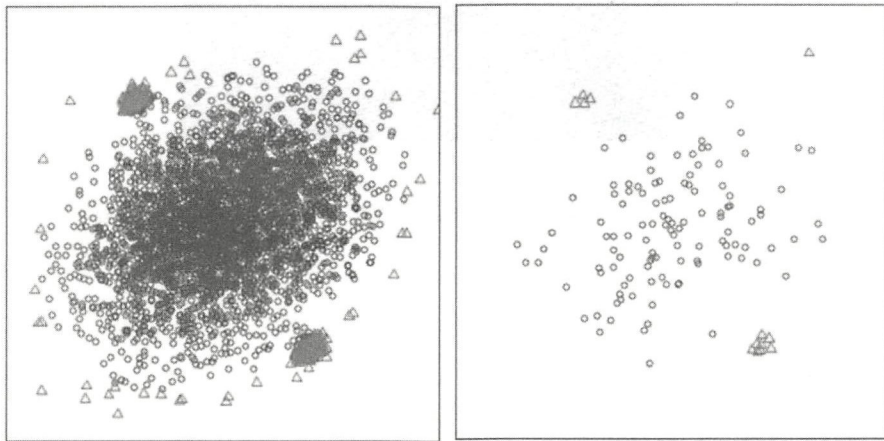


图11-5 数据分布图（左图：采样前；右图：采样后）

iForest 构建好之后，同样计算记录的异常指数 $S(x, n)$ ：

$$S(x, n) = 2^{\left(-\frac{E(h(x))}{c(n)}\right)} \quad (11-5)$$

其中， $S(x, n)$ 就是记录 x 在由 n 个样本的训练数据构成的 iForest 的异常值得分， $E(h(x))$ 表示记录 x 在每棵树的高度均值， $c(n)$ 的定义同公式 (11-4)。

综上所述，独立森林本质上是一种非监督算法，不需要先验的类标签。在处理高维数据时，不是把所有的属性都用上，而是通过峰度系数挑选一些有价值的属性，然后再进行 iForest 的构造，算法效果会更好。

接下来，我们通过一个示例（Python 实现代码见下文）利用 iForest 算法进行异常点检测。首先，有一个训练集 train，用其训练 iForest 模型。然后，输入 test 数据集和 outlier 数据集，得到模型预测结果 (y_pred_test, y_pred_outlier)，1 代表预测为正常点，-1 代表预测为异常点。最

后，我们对整个图形空间的点输入模型，得到如图 11-6 所示的结果，颜色越深代表异常值得分越高，颜色越浅代表异常值得分越低。可以看出，模型预测效果不错，颜色分布规律与 test 和 outlier 数据集完全吻合。

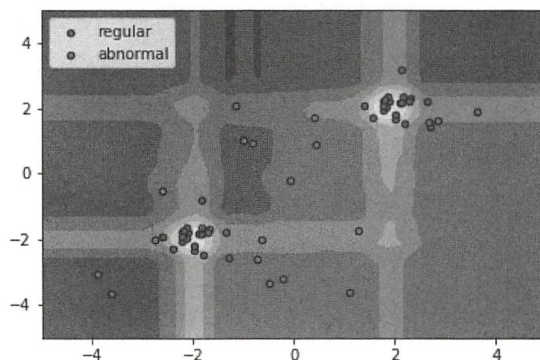


图11-6 iForest算法预测结果

##独立森林示例实现代码

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest

rng = np.random.RandomState(42)

# 生成训练数据
X = 0.3 * rng.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# Generate some regular novel observations
X = 0.3 * rng.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# Generate some abnormal novel observations
X_outliers = rng.uniform(low=-4, high=4, size=(20, 2))

plt.scatter(X_outliers.T[0], X_outliers.T[1])
plt.show()

# fit the model
clf = IsolationForest(max_samples=100, random_state=rng)
clf.fit(X_train)
y_pred_train = clf.predict(X_train)
```

```
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)

# plot the line, the samples, and the nearest vectors to the plane
xx, yy = np.meshgrid(np.linspace(-5, 5, 50), np.linspace(-5, 5, 50))
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Blues_r)
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='green',
                 s=20, edgecolor='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='red',
               s=20, edgecolor='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([b2, c], ["regular", "abnormal"], loc="upper left")
plt.show()
```

11.4 本章小结

异常检测是很多具有故障应用的实时监控系统的核心部分，也被应用到非常广泛的领域，如检测、欺诈检测、网络入侵检测等。尽管它至关重要，但是实际上实现全自动异常检测系统是一项具有挑战性的任务，这些挑战通常会导致解决方案不是可扩展的或高度专业化的，也导致较高的误报率。

11.5 参考文献

- [1] Yahoo.Generic and Scalable Framework for Automated Time-series Anomaly Detection
- [2] Pang-Ning Tan, Michael Steinbach, Vipin Kumar 著. 范明, 范宏建等译. 数据挖掘导论. 完整版. 北京: 人民邮电出版社, 2011 年
- [3] 王斌会, 陈一非. 基于稳健马氏距离的多元异常值检测. 统计与决策, 3x(2005): 4-6
- [4] F. T. Liu, M. T. Kai, Z. H. Zhou. Isolation Forest. Eighth IEEE International Conference on Data Mining IEEE, 2009: 413-422

第 12 章

故障诊断和分析策略

根据 IBM 沃森研究中心的论文 *A survey of fault localization techniques in computer networks*^[1] 中的定义，故障管理领域的基本概念如下。

事件：定义为软硬件或操作中发生的异常情况。

故障（也称为问题或根源）：一类可能导致其他事件发生，但不是由其他事件引起的事件。根据持续时间故障可以分为如下几种。

- **永久的：**在修复之前会一直存在。
- **间歇性的：**发生在不连续和周期性基础上的短时间的服务退化。
- **短暂的：**瞬时的故障会导致服务暂时性轻微退化，它们通常会自动恢复（通过自动降级、扩容、摘除等自动化手段）。

错误：计算值、观测值与真实的或理论上的正确值之间的差异，错误是故障的后果，故障可能会导致一个或多个错误。

症状：失败的外在表现，一般会通过告警、Dashboard 等被观察到。

在前面的章节中已经涉及了部分告警聚合和收敛、异常点检测等和故障分析相关的技术，正如读者所看到的，除了上面介绍的智能化辅助方法，故障诊断和定位还需要一些基础工作的支持。

一般情况下，业务系统经过多年的演化，各个环节、不同部门之间已经不太可能采用同一套开发语言、框架、容器和流程，因此，每个系统记录的日志格式、发出的告警、自动化手段也是各不相同、纷繁复杂的，很难做到统一标准。

基于这个前提（异构、无统一标准），一些介入成本较小（不要求开发和运维人员改变技术栈）、效果又明显的基础工作，可以大大提高异构系统之间的关联性，提供相对规范的数据。

12.1 日志标准化

在异构系统的互联互通中，日志是第二容易做到标准化的（第一是接口）。日志标准化的含义就是：

- 日志的内容要可理解（是指机器可理解）。
- 格式要相对统一。
- 能自我标识（说明自己的来源，如业务线、服务等）。

日志标准化的好处是显而易见的。首先，标准化的日志有利于提高自动化程度。在前面的章节中读者已经了解到日志的采集、传输、ETL、运算环节，非标准化的日志（很遗憾，现实中大部分日志都是这样的）会加重后面的 ETL、运算环节的负担，每一类日志都要对应单独的 ETL 过程和运算模型，在运维实践中，日志的 ETL 和运算通常都是付出成本最多的地方。如果日志相对有标准，那么后面的各个环节的通用性和自动化程度就会提高不少。比如，访问日志大家都按照 Nginx/Apache 格式来规范、记录，那么后面的 ETL 和运算模型可以大大简化，新接入的日志就可以自动处理，不必为每一类日志都单独做 ETL。

其次，如果日志能实现自我标识，那么会大大方便对故障的分析和定位。举一个反例：有些部门前端用的是 PHP，后端用的也是 PHP，这样通过 PHP 日志函数记录的日志看起来完全一样，没有标识前后端信息，如果集中处理这些日志，除非在采集/传输过程中增加标识（大部分采集/传输程序都支持给日志加标签），否则这样的日志在分析和定位故障工作中很难有比较大的帮助。

在运维实践中，很多时候为了提高日志采集、接收的效率，而不做任何标准化和预处理，“先收上来再说”，导致大量冗余、无用的日志信息占用空间和带宽，长时间没人使用。所以，在日志收集之前，需要付出一点成本，做一些标准化和规范化的工作是有必要的。

另外，需要说明的是，日志标准化指的是日志格式和内容要有一定的标准。它不同于“数据标准化”这个概念，数据标准化是为了对数据进行分析，对数据进行指数化，将数据按比例缩放，使之落入一个小的特定区间，以便不同单位和量级的数据可以进行比较和加权。

12.2 全链路追踪

一提到全链路追踪，大家一定会想到基于 Google Dapper 思想的各种系统（比如 Twitter 的 Zipkin、阿里巴巴的鹰眼等），具体实现到什么程度，主要看投入程度和协作能力，毕竟对前后端还是有一定的改造成本的，还可能牵扯到跨部门合作。

相对于可能实现到函数级别追踪的 Trace 系统，基于 RequestID 的分布耗时统计成本要少很多，甚至可以只针对容器级别进行统计，定位到调用链上的某个步骤，一般情况下应该够用了。

在具体实践中，我们常常利用 Nginx 的 \$request_id 变量来唯一标识一次请求。通过以下指令来完成标识：

```
proxy_set_header HTTP_REQUEST_ID $request_id;
```

可以将本次请求的 request_id 放在 Header 中传到后端，后面的 Web 容器、Web 应用等通过 HTTP 环境变量 HTTP_REQUEST_ID 即可获取，这样在记录相关日志时，即可使用此 ID，达到串联前后端、调用链的目的。

有了 request_id，还不能绘制出一个完整的调用链路图，还缺少一个元素来指明上下文和嵌套关系，有些系统使用字符串 ID（0, 0.1, 0.1.1）来说明访问顺序。在业务场景不复杂的情况下，可以手工维护上下文关系，但是在大型系统中，这种上下文关系可能需要专门维护、自动关联。

12.3 SLA的统一

在分布式系统中，尤其是当不同部门开发的业务系统需要相互协作时，如果没有统一、无歧义的 SLA 约定，你可以想象一下以下场景：

A 部门调用 B 部门提供的服务，为了监测服务的可用性，A 部门用返回错误码 5xx 的比例来评估服务，一旦 5xx 数量大于阈值，就会当作故障通报给 B 部门。然而，B 部门作为服务提供者，使用“响应时间”作为服务可用性指标，只要总体响应时间不超过对外承诺的 SLA 约定，服务就没有问题（5xx 可能由多种因素导致）。由于两边标准不一致，给故障的及时排查、定位带来了很大的困难。

尤其是当你希望通过自动化、智能化的方法来定位和排查故障时，SLA 的统一可以让这些工作更加容易。

12.4 传统的故障定位方法

在前面的章节中，我们介绍了告警的收敛和聚合，告警信息经过收敛和概化，必然会损失一些细节信息，对于业务负责人来说，其只需关心现在服务是否正常、怎么恢复，而对具体是哪个点不正常、为什么不正常并不特别关心；但对于运维人员来说，除必要的快速恢复、止损、降级等掩盖手段外，其恰恰还需要关心具体的故障点、故障原因，以便彻底优化和根除。

传统的故障定位手段有监控告警型和日志分析型等。

12.4.1 监控告警型

理论上，只要监控做得足够全，所有问题都应该能被“发现”。发现问题的流程一般是：收到告警→查看 Dashboard→缩小范围→定位故障点。告警和 Dashboard 组织得比较好的话，对于一般的问题，通过这样的方式还能应付过来。图 12-1 展示了通过监控告警定位故障的流程。

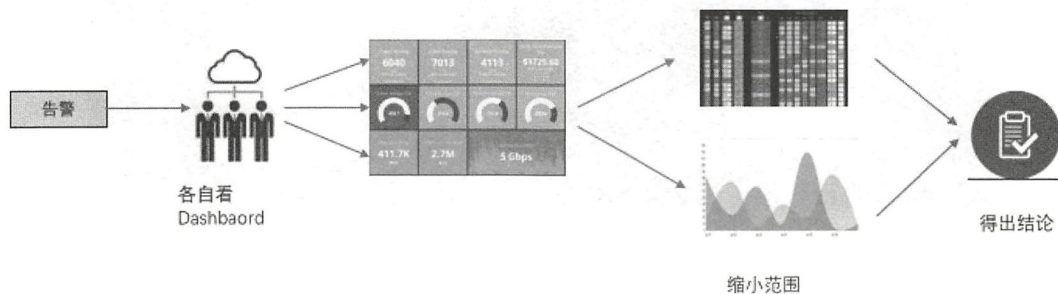


图12-1 通过监控告警定位故障的流程图

这种方式虽然有效，但是有以下几个缺点。

- 非常依赖经验。随着人员的流动、系统的迭代，真正全面掌控每个细节的人越来越少，培养这样的人成本较高，新人上手困难。
- 系统规模越大，定位成本越高。系统规模大意味着数据量、Dashboard、告警量也会变大，但运维工程师的数量却是有限的。
- 实时性不够。在上班时间还勉强能及时响应，而在非工作时间，能把了解情况的人都找齐就不容易了。
- 不够精确。只能知道大概出问题的点，但是具体原因可能需要上服务器具体分析。

12.4.2 日志分析型

前面介绍过日志标准化、增加 request_id 等内容，基于日志分析的问题定位技术，明显的好处就是包含了所有细节，问题出现过就会留下日志，所以相对要准确得多。通过一些离线分析技术（比如 Logstash+Hadoop+Hive、ELK、Pandas 等），再加上 request_id，可以将单个请求的链路访问过程展示出来，方便对问题进行定位。

另外，也可以对日志中的关键字、错误码/状态码进行一些实时的统计和汇总，这样能比较直观地反映问题出现的趋势，如图 12-2 和图 12-3 所示。

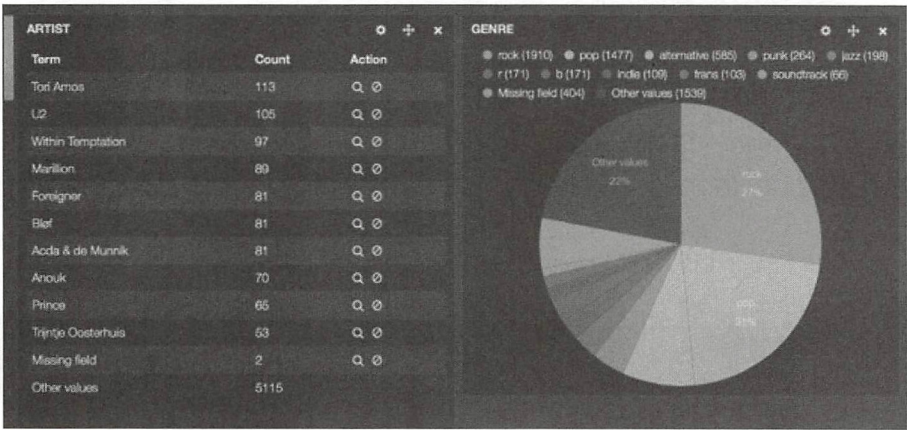


图12-2 日志中的状态码统计



图12-3 Elasticsearch的链路日志分析

基于日志分析的问题定位技术的局限性也很明显，如下所示。

- 当系统规模比较大时，日志无法被全部收集。由于成本问题，日志分析作为辅助系统，规模一般大大小于线上业务系统，注定不可能收集线上所有的日志，只能采样或者收集某一类日志（比如错误日志），那么分析抽样日志得出的结论不一定能反映出真实的状况。
- 不够及时。这类分析需要人工参与的部分比较多，一般用于事后追溯比较好。
- 极度依赖人的经验。

12.5 人工智能在故障定位领域的应用

在前面的章节中，我们已经讨论了故障诊断相关内容（如异常检测、多维数据聚合和关联、告警收敛等），下面我们介绍故障定位相关技术。

前面提到传统的故障定位技术（监报告警型和日志分析型）非常依赖人的经验，但是，一方面，经验的传承与积累受组织结构和人事变动的影响非常大；另一方面，这些代表经验或知识的规则显然是隐含的，只可意会不可言传，很难被人为地总结成基于显式规则的专家系统。人工智能就是通过算法和不断学习，实现隐含规则的自动学习以及更高知识粒度的学习推理。

12.5.1 基于关联规则的相关性分析

就像本章开头所介绍的，故障一般会通过事件、错误、症状表现出来，前面我们也介绍过通过聚类来处理告警（降低维度），那么这些事件、错误、告警数据之间有没有联系？它们之间的关系是否能为故障定位提供帮助？

“关联规则挖掘是指从一个大型的数据集中发现有趣的关联或相关关系，即从数据集中识别出频繁出现的属性值集（也称为频繁项集），然后利用这些频繁项集创建描述关联关系规则的过程。”^[2]

关联规则挖掘是一种基于规则的机器学习算法，它的目的是利用一些度量指标来分辨数据集中存在的强规则。也就是说，关联规则挖掘用于知识发现，而非预测，所以属于无监督的机器学习算法。关联规则的度量（筛选和过滤）方法有：最小支持度（Minimum support）和最小置信度（Minimum confidence）。

下面我们通过数据挖掘领域著名的“啤酒与尿布”^[3]案例进行介绍，如图 12-4 所示。

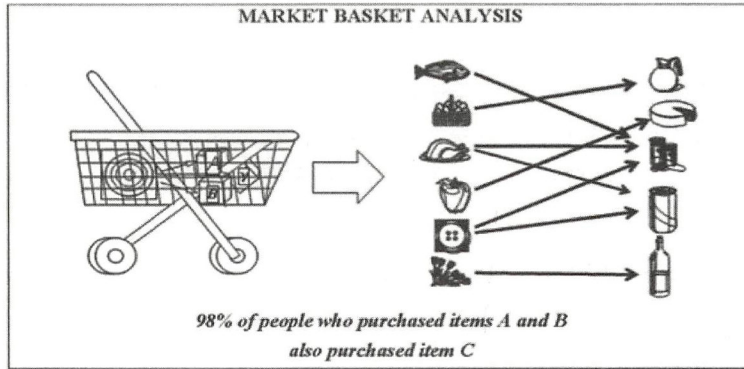


图12-4 啤酒与尿布的购物篮分析案例

图 12-5 展示的是超市购物篮的规则挖掘，买了 Item A 和 Item B 的人，也买了 Item C，这就是一条规则： $\{A,B\} \rightarrow \{C\}$ 。下面我们把购物篮变成数据集。

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Rules Discovered:
 $\{Milk\} \rightarrow \{Coke\}$
 $\{Diaper, Milk\} \rightarrow \{Beer\}$

图12-5 关联规则发现示例

从图 12-5 可以看到， $\{Milk\} \rightarrow \{Coke\}$ 和 $\{Diaper, Milk\} \rightarrow \{Beer\}$ 就是发现的两条规则：第一条，买牛奶的人同时会买可乐；第二条，买尿布和牛奶的人同时会买啤酒。当然，这个样本有点少，在实际应用中数据量会很大。

关联规则就是有关联的规则，形式是这样定义的：两个不相交的非空集合 X 、 Y ，如果有 $X \rightarrow Y$ ，就说 $X \rightarrow Y$ 是一条关联规则。举个例子，在图 12-5 所示的表中，我们发现购买了牛奶就一定会购买可乐， $\{牛奶\} \rightarrow \{可乐\}$ 就是一条关联规则。关联规则的强度用支持度（support）和置信度（confidence）来描述。

支持度的定义： $support(X \rightarrow Y) = |X \cap Y| / N$ = 集合 X 与集合 Y 中的项在一条记录中同时出现的次数/数据记录的个数。例如： $support(\{牛奶\} \rightarrow \{可乐\}) = 牛奶和可乐同时出现的次数/数据记录数 = 3/5 = 60\%$ 。

置信度的定义： $confidence(X \rightarrow Y) = |X \cap Y| / |X|$ = 集合 X 与集合 Y 中的项在一条记录中同时出现的次数/集合 X 出现的个数。例如： $confidence(\{牛奶\} \rightarrow \{可乐\}) = 牛奶和可乐同时出现的$

次数/可乐出现的次数=3/3=100%； $\text{confidence}(\{\text{可乐}\} \rightarrow \{\text{牛奶}\}) = \text{可乐和牛奶同时出现的次数} / \text{牛奶出现的次数} = 3/4 = 75\%$ 。

这里定义的支持度和置信度都是相对的，不是绝对的，绝对的支持度 $\text{abs_support} = \text{数据记录数 } N \times \text{support}$ 。支持度和置信度越高，说明规则越强，关联规则挖掘就是挖掘出满足一定强度的规则。

关联规则挖掘所花费的时间主要在生成频繁项集上，因为找出的频繁项集往往不会很多，利用频繁项集生成规则也就不会花太多的时间；而生成频繁项集需要测试很多的候选项集，如果不加以优化，所需的时间是 $O(2^N)$ 。

关联规则的挖掘过程（Apriori 算法）如下：

- 生成频繁项集。挖掘频繁项集，根据最小支持度阈值（ min_sup ）找出数据集 DB 中的所有频繁项集，直到再没有满足 min_sup 条件的项集为止。
- 利用频繁项集构造出满足用户最小置信度的规则。

具体做法如下：

首先找出频繁 1-项集，记为 L_1 ；然后利用 L_1 产生候选项集 C_2 ，对 C_2 中的项进行判定挖掘出 L_2 ，即频繁 2-项集；如此循环下去，直到无法发现更多的频繁 k -项集为止。每挖掘一层 L_k 就需要扫描整个数据库一遍。Apriori 算法挖掘过程如图 12-6 所示。

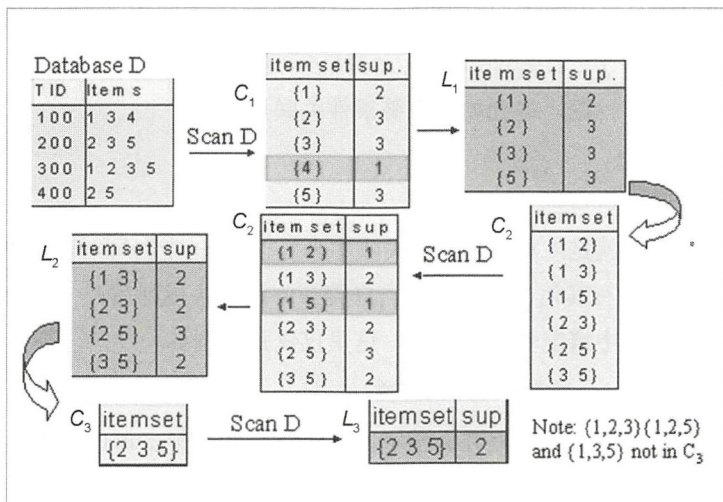


图12-6 Apriori算法挖掘过程

说明：

(1) 连接： $C_3=L_2$, $L_2= \{\{A,C\},\{B,C\},\{B,E\}\{C,E\}\} \{\{A,C\},\{B,C\},\{B,E\}\{C,E\}\} = \{\{A,B,C\},\{A,C,E\},\{B,C,E\}\}$

(2) 使用 Apriori 性质剪枝：频繁项集的所有子集必须是频繁的，对候选项集 C_3 ，我们可以删除其子集为非频繁的选项：

$\{A,B,C\}$ 的 2-项子集是 $\{A,B\},\{A,C\},\{B,C\}$ ，其中 $\{A,B\}$ 不是 L_2 的元素，所以删除这个选项； $\{A,C,E\}$ 的 2-项子集是 $\{A,C\},\{A,E\},\{C,E\}$ ，其中 $\{A,E\}$ 不是 L_2 的元素，所以删除这个选项； $\{B,C,E\}$ 的 2-项子集是 $\{B,C\},\{B,E\},\{C,E\}$ ，它的所有 2-项子集都是 L_2 的元素，因此保留这个选项。

(3) 这样，剪枝后得到 $C_3=\{\{B,C,E\}\}$ 。

在上一步产生的频繁项集的基础上生成满足最小置信度的规则，就称为强规则。

Apriori 算法的性能瓶颈：需要产生大量的候选项集，以及需要重复地扫描数据库。2000 年 Jiawei Han 等人提出了基于 FP 树生成频繁项集的 FP-growth 算法。该算法只进行两次数据库扫描且不使用候选项集，直接将数据库压缩成一个频繁模式树，最后通过这棵树生成关联规则。研究表明，它比 Apriori 算法大约快一个数量级。

当关联规则生成后，可以用来辅助进行事件（错误、告警）相关性分析。当一个告警/事件发生时，用关联规则进行匹配，可以判断是独立的告警/事件，还是关联的告警事件，从而确定故障的根源，定位故障。

现在我们通过分析和挖掘，得到如下告警事件关联规则：

```
{HBase 负载高} → {对象库延迟}
{对象库延迟} → {发微博功能失败率高}
```

假如现在故障的症状就是“发微博功能失败率高”，那么一定会同时收到“对象库延迟”和“HBase 负载高”（或许还有其他）的告警。当告警命中这两条规则后，一个简单的故障根源就可以被追踪到了（在实践中情况要复杂得多，这里仅仅是示例）：

```
{HBase 负载高} → {对象库延迟} → {发微博功能失败率高}
```

由于篇幅有限，这里就不深入展开介绍了。关于算法的实现，可以参考 <https://github.com/asaini/Apriori>。

12.5.2 基于决策树的故障诊断

决策树^{[5][6]}是附加概率结果的一个树状的决策图，是直观地运用统计概率分析的图法。在机器学习中，决策树是一个预测模型，表示对象属性和对象值之间的一种映射，树中的每一个节点都表示对象属性的判断条件，其分支表示符合节点条件的对象。树的叶子节点表示对象所属的预测结果。

图 12-7 展示了一个简单的决策树模型。

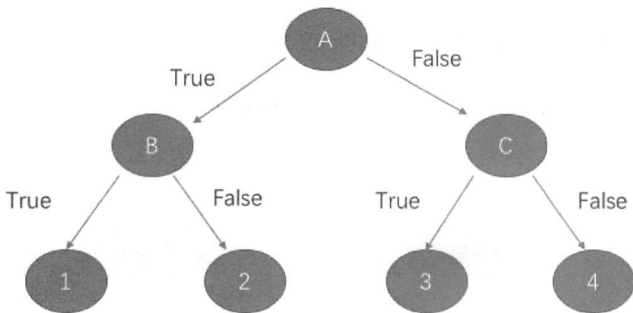


图12-7 简单的决策树模型

和关联规则不同，决策树属于有监督的机器学习方式。在实际应用中，可以通过对数据进行处理，利用归纳算法生成可读的规则和决策树，然后使用决策树对新数据进行分析。决策树是一种非常典型分类方法（算法）。

简单来说，决策树算法其实就是根据已有的经验来构建一棵树。可以认为是根据数据的某个维度进行切分的，并不断重复这个过程。当然，如果切分的顺序不同，则会得到不同的树。

回到故障分析和诊断问题上，假如对于某类故障的处理有了一定的经验，形成了一系列的规则，就可以对这些规则进行总结，当某个应用发生告警时，其判断逻辑可以取如图 12-8 所示的经验值。

AvgTime超过阈值：是	容量是否足够：是	是否单机问题：是	单机503
AvgTime超过阈值：是	容量是否足够：是	是否单机问题：否	降级
AvgTime超过阈值：是	容量是否足够：否	是否单机问题：是	先503后扩容
AvgTime超过阈值：是	容量是否足够：否	是否单机问题：否	扩容
AvgTime超过阈值：否	容量是否足够：是	是否单机问题：是	单机503
AvgTime超过阈值：否	容量是否足够：是	是否单机问题：否	需要其他辅助信息
AvgTime超过阈值：否	容量是否足够：否	是否单机问题：是	先503后扩容
AvgTime超过阈值：否	容量是否足够：否	是否单机问题：否	扩容

图12-8 故障分析中的经验总结

这些判断过程可以被绘制成一棵树，如图 12-9 所示。

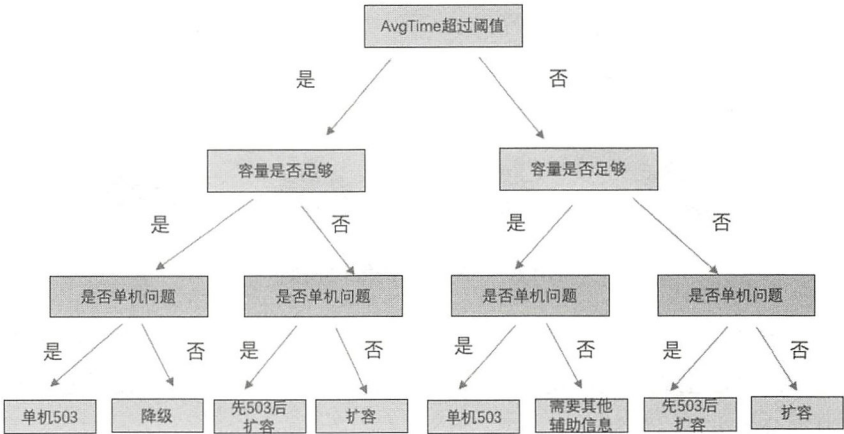


图12-9 基于AvgTime超过阈值的决策树

这就是决策树，在每一层我们都提出了一个问题，然后根据问题的回答走向不同的子树，最终到达叶子节点时做出决策（具体操作）。当然，在实践中，维度会比这里多很多。

决策树被用在故障诊断中有如下几个明显的优势。

- 根据人的经验来构建决策树，易于理解和实现。
- 效率高。决策树只需构建一次，可以反复使用，每一次预测的最大计算次数不超过决策树的深度。
- 对中间值的缺失不敏感。
- 数据简单，并且不需要规范化。

构建决策树通常采用自上而下的方法，每一步都选择一个最好的属性来分裂。“最好”的定义是使得子节点中的训练集尽量纯。不同的算法使用不同的指标来定义“最好”。

决策树的构建算法有如下几种。

1. ID3 (迭代二叉树 3 代): 采用信息增益来选择树杈

信息增益的含义就是指划分数据前后信息发生的变化。信息增益的思想来源于信息论的香农定理 (信息熵)。ID3 算法选择具有最高信息增益的自变量作为当前的树杈 (树的分支)，其核心思想是自顶向下贪婪搜索遍历可能的决策树空间构造决策树，以信息增益度量属性的选择，选择分裂后信息增益最大的属性进行分裂，即递归选择分类能力最强的特征对数据进行分割。

信息熵表示的是不确定度。在均匀分布时，不确定度最大，此时熵就最大。当选择某个特征对数据集进行分类时，分类后的数据集信息熵会比分类前小，其差值表示为信息增益。信息增益可以衡量某个特征对分类结果的影响大小。

信息熵公式如下：

$$H(D) = -\sum_{i=0}^k p_i \log p_i \quad (12-1)$$

其中 D 为样本集合。

条件熵公式如下：

$$H(Y|X) = \sum_{i=0}^n p_i H(Y|X = x_i) \quad (12-2)$$

在决策树算法中，划分前样本集合 D 的熵是一定的，为 $H(D)$ ，使用某个特征 A 划分数据集 D ，计算划分后的数据子集的熵，为 $H(D|A)$ ，则信息增益为划分前后的熵值之差。公式如下：

$$I(D, A) = H(D) - H(D|A) \quad (12-3)$$

ID3 决策树算法就用到了上面的信息增益，每次分裂都贪心选择信息增益最大的属性，作为本次分裂属性。每次分裂都会使树长高一层。这样逐步做下去，就可以构建一棵决策树。

2. C4.5: 采用增益率

ID3 有一些缺陷，就是在分类时容易选择一些比较容易分裂的属性，尤其在具有像 ID 值这样的属性，因为每个 ID 都对应一个类别，所以会造成分类很细碎。

C4.5 算法定义了分裂信息，公式如下：

$$\text{split_info}_A(D) = -\sum_{j=1}^v \frac{|D_j|}{|D|} \log_2 \left(\frac{|D_j|}{|D|} \right) \quad (12-4)$$

很容易理解，这也是一个熵的定义， $p_i = \frac{|D_i|}{|D|}$ ，可以看作是属性分裂的熵，分得越多就越混乱，熵就越大。定义信息增益率，公式如下：

$$\text{gain_ratio}(A) = \frac{\text{gain}(A)}{\text{split_info}(A)} \quad (12-5)$$

C4.5 就是选择最大增益率的属性来分裂，其他类似于 ID3.5。

3. CART (分类回归树)：采用Gini指数

Gini 指数表示在样本集合中一个随机选中的样本被分错的概率。

- 它是一种不等性度量。
- 在经济学中它通常被用来度量收入的不平衡度，可以推广到更广泛的场景，用于度量任何不均匀分布。
- 它是介于 0~1 之间的数，0—完全相等，1—完全不相等。
- 总体包含的类别越杂乱，Gini 指数就越大（跟熵的概念很相似）。

Gini 指数定义如下：

$$\text{Gini}(p) = \sum_{k=1}^K p_k(1 - p_k) = 1 - \sum_{k=1}^K p_k^2 \quad (12-6)$$

这个公式的特点是：Gini 指数是统计学上的抽象概念，它有明确定义的科学名词且与内容无关；不随信息的具体表达式的变化而变化；独立于形式，反映了信息表达式中统计方面的性质。

ID3 的 Python 实现：请参见 <https://github.com/NinjaSteph/DecisionTree>。

C4.5 的 Python 实现：请参见 <https://github.com/geerk/C45algorithm>。

决策树的生成步骤如下：

- (1) 收集数据。
- (2) 处理数据，对数据进行预处理和清洗加工。

(3) 对多个可以划分的特征进行甄选。甄选的标准是保证通过甄选之后划分的结果的信息增益最大。划分方式是按照当前甄选的特征进行划分。

(4) 如果划分后的数据全是同一种类型（即香农熵为 0），则不再划分；否则，继续递归调用第 3 步的方法。

下面介绍一下决策树的剪枝。

决策树为什么要剪枝？原因就是避免决策树“过拟合”样本。通过前面的算法生成的决策树非常详细而庞大，每个属性都被详细考虑了，决策树的树叶节点所覆盖的训练样本都是“纯”的。因此，用这棵决策树对训练样本进行分类的话，你会发现对于训练样本而言，这棵树的表现堪称完美，它可以 100% 完美正确地对训练样本集中的样本进行分类（因为决策树本身就是 100% 完美拟合训练样本的产物）。

但是，这会带来一个问题。如果训练样本中包含了一些错误，按照前面的算法，这些错误也会 100% 一点不留地被决策树学习，这就是“过拟合”。C4.5 的缔造者昆兰教授很早就发现了这个问题，他做过一个试验，得到的结果是：在某一个数据集中，过拟合的决策树的错误率比一个经过简化了的决策树的错误率要高。那么现在的问题就是，如何在原生过拟合的决策树的基础上，通过剪枝生成一个简化了的决策树呢？

（1）预剪枝（Pre-Pruning）

预剪枝是指在构建决策树的同时进行剪枝。所有决策树的构建方法，都是在无法进一步降低熵的情况下才会停止构建分支的过程的。为了避免过拟合，可以设定一个阈值，当熵减小的数量小于这个阈值时，即使还可以继续降低熵，也停止继续构建分支。但是在实际中，这种方法的效果并不好。

（2）后剪枝（Post-Pruning）

后剪枝是指在决策树构建完成后进行剪枝。剪枝的过程是对拥有同样父节点的一组节点进行检查，判断如果将其合并，熵的增加量是否小于某一阈值。如果确实小，则这一组节点可以合并为一个节点，其中包含了所有可能的结果。后剪枝是目前最普遍的做法。后剪枝的过程是删除一些子树，然后用其叶子节点代替，这些叶子节点所标识的类别通过大多数原则（majority class criterion）确定。所谓大多数原则，是指在剪枝过程中，将一些子树删除而用一个叶子节点代替，而这个叶子节点所属的类别，可以用这棵子树中大多数训练样本所属的类别来进行标识，所标识的类称为 majority class。

12.6 本章小结

故障诊断和分析是智能运维研究的非常重要的一个领域，高效地进行故障诊断和分析可以提高系统的可用性，一个好的诊断方法能够快速、高效地找到故障根源，加快解决问题，在一定程度上降低了故障的持续时间，减小了因故障带来的损失。

在智能运维领域存在三种场景：历史事件分析、当前事件分析和未来事件分析。故障诊断是针对已经发生或正在发生的事件进行分析的。本章重点讨论了故障诊断和分析的方法，包括传统方法和基于人工智能的分析方法。在实际应用中，我们需要结合具体业务，采用不同的方法，以达到最好的分析效果。

12.7 参考文献

- [1] Malgorazta Steinder, Adarshpal S.Sethi. A survey of fault localization techniques in computer networks, 2004
- [2] 李金凤, 王怀彬. 基于关联规则的网络故障告警相关性分析, 2012
- [3] 啤酒与尿布（数据挖掘领域著名案例，用来解释关联规则的提取）
- [4] Papia Ray, Debani Prasad Mishra. Support vector machine based fault classification and location of a long transmission line, 2015
- [5] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, Eric Brewer. Failure Diagnosis Using Decision Trees, 2004
- [6] Ngoc-Tu Nguyen, Jeong-Min Kwon, Hong-Hee Lee. A Study on Machine Fault Diagnosis using Decision Tree, 2007

第13章

趋势预测算法

在实时监控系统中所处理的数据通常具有时间特征，因此被称作时间序列，也叫作时间数据或者动态数列。它是将某种统计指标的数值，按时间先后顺序排列所形成的数列。时间序列预测法就是通过编制和分析时间序列，根据时间序列所反映出来的发展过程、方向和趋势，进行类推或延伸，借以预测下一段时间或以后若干年内可能达到的水平。其内容包括：收集与整理某种社会现象的历史资料；对这些资料进行检查鉴别，排成数列；分析时间数列，从中寻找该社会现象随时间变化的规律，得出一定的模式；以此模式来预测该社会现象将来的情况，比如预测某产品的请求量（PV）和用户增长趋势。

时间序列一般具有如下比较明显的特征。

- 趋势性（Trend）：数据呈现某种持续向上或向下的趋势或者规律。
- 季节性（Seasonality）：数据呈现季节性，数据取值重复出现的现象。
- 周期性（Cyclical）：也称循环波动性或者周期波动现象。
- 随机性（Random）：也称不规则波动。

时间序列预测法可用于短期、中期和长期预测。在进行非平稳时间序列（Non-stationary series）分析时，若导致非平稳的原因是确定的，则可以使用的方法主要有趋势拟合模型、季节调整模型、移动平均法、指数平滑法等；若导致非平稳的原因是随机的，则方法主要有ARIMA模型和自回归条件异方差模型等。下面将详细介绍移动平均法、指数平滑法、ARIMA模型和神经网络模型几种方法。

13.1 移动平均法

移动平均法是一种简单的平滑预测技术。它的基本思想是基于时间序列的历史数据，依次

计算移动周期的时序数据平均值，用来预测未来的发展趋势。因此，其优点是计算简单，只需取历史数据的平均数即可。但缺点也显而易见，若时序数据的波动比较剧烈，采用平均数的方法会使很多趋势特征丢失，从而得不到好的预测结果。

移动平均法可以分为简单移动平均和加权移动平均。简单移动平均的各元素的权重相等。预测公式如下：

$$S_t = \frac{(\alpha_{t-1} + \alpha_{t-2} + \cdots + \alpha_{t-n})}{n} \quad (13-1)$$

其中， S_t 为 t 期的预测值； n 为移动平均的时期个数； α_{t-n} 为 $t-n$ 期实际值。

加权移动平均给不同时期的变量值赋予不同的权重。其基本原则是对近期的变量值赋予较高的权重，对远离目标期的变量值赋予较低的权重。预测公式如下：

$$S_t = \frac{(\alpha_{t-1}W_1 + \alpha_{t-2}W_2 + \cdots + \alpha_{t-n}W_n)}{n} \quad (13-2)$$

其中， S_t 为 t 期的预测值； W_1 为第 $t-1$ 期实际值的权重； W_n 为第 $t-n$ 期实际值的权重； n 为预测的时期数； $W_1 + W_2 + \cdots + W_n = 1$ 。

在加权移动平均法中，对权重的选择是一个难题，一般通过尝试和经验来决定。

13.2 指数平滑法

指数平滑法是一种特殊的加权移动平均法。其优势在于兼容了全期平均和移动平均所长，不舍弃过去的的数据，仅给予逐渐减弱的影响程度，即随着数据的远离，赋予逐渐收敛为 0 的权重数，这也避免了在加权移动平均法中需要指定每一个参数权重的劣势。

一般常用的指数平滑法为一次指数平滑、二次指数平滑和三次指数平滑。高次指数平滑一般比较难见到，因此我们重点介绍一次、二次和三次指数平滑。

在平滑之前，首先需要有初始值。通常，如果数据项小于 20，则初始值取第 1、2、3 期的平均值；如果数据项大于 20，则可以把第 1 期的值作为初始值，计作 S_0 。

(1) 一次指数平滑一般应用于水平型数据。其计算公式为：

$$S_t^{(1)} = \alpha x_{t-1} + (1 - \alpha) S_{t-1}^{(1)} \quad (13-3)$$

其中, $S_t^{(1)}$ 为 t 期的平滑值, $S_{t-1}^{(1)}$ 为 $t-1$ 期的平滑值, x_{t-1} 为 $t-1$ 期的实际值, α 为常数。

预测未来期的公式为:

$$S_{t+1}^{(1)} = \alpha x_t + (1 - \alpha) S_t^{(1)} \quad (13-4)$$

其中, 对 α 的选择是一个难题, 一般也是通过经验法。当数据波动不大时, 选择较小的 α 值; 当数据波动较大时, 选择较大的 α 值。

(2) 二次指数平滑适用于斜坡型时序数据, 其基本思想是在一次平滑之后, 再进行一次平滑。其计算公式为:

$$\begin{aligned} S_t^{(1)} &= \alpha x_t + (1 - \alpha) S_{t-1}^{(1)} \\ S_t^{(2)} &= \alpha S_t^{(1)} + (1 - \alpha) S_{t-1}^{(2)} \end{aligned} \quad (13-5)$$

其中, $S_t^{(1)}$ 为 t 期的一次平滑值, $S_t^{(2)}$ 为 t 期的二次平滑值, $S_{t-1}^{(1)}$ 为 $t-1$ 期的一次平滑值, $S_{t-1}^{(2)}$ 为 $t-1$ 期的二次平滑值, x_t 为 t 期的实际值, α 为常数。

预测未来 T 期的计算公式为:

$$x_{t+T} = A_T + B_T T \quad (13-6)$$

其中

$$\begin{aligned} A_T &= 2S_t^{(1)} - S_t^{(2)} \\ B_T &= \frac{1}{1 - \alpha} (S_t^{(1)} - S_t^{(2)}) \end{aligned}$$

(3) 三次指数平滑则是在二次平滑的基础上再平滑, 适用于同时含有趋势性和季节性的序列数据。其计算公式如下:

$$\begin{aligned} S_t^{(1)} &= \alpha x_t + (1 - \alpha) S_{t-1}^{(1)} \\ S_t^{(2)} &= \alpha S_t^{(1)} + (1 - \alpha) S_{t-1}^{(2)} \\ S_t^{(3)} &= \alpha S_t^{(2)} + (1 - \alpha) S_{t-1}^{(3)} \end{aligned} \quad (13-7)$$

其中, $S_t^{(1)}$ 为 t 期的一次平滑值, $S_t^{(2)}$ 为 t 期的二次平滑值, $S_t^{(3)}$ 为 t 期的三次平滑值, $S_{t-1}^{(1)}$ 为 $t-1$ 期的一次平滑值, $S_{t-1}^{(2)}$ 为 $t-1$ 期的二次平滑值, $S_{t-1}^{(3)}$ 为 $t-1$ 期的三次平滑值, x_t 为 t 期的实际值, α 为常数。

预测未来 T 期的计算公式为：

$$x_{t+T} = A_T + B_T T + C_T T^2 \quad (13-8)$$

其中

$$A_T = 3S_t^{(1)} - 3S_t^{(2)} + S_t^{(3)}$$

$$B_T = \frac{\alpha}{2(1-\alpha)^2} [(6-5\alpha)S_t^{(1)} - 2(5-4\alpha)S_t^{(2)} + (4-3\alpha)S_t^{(3)}]$$

$$C_T = \frac{\alpha^2}{2(1-\alpha)^2} [S_t^{(1)} - 2S_t^{(2)} + S_t^{(3)}]$$

13.3 ARIMA 模型

13.3.1 简介

ARIMA 模型的全称为自回归移动平均模型（Auto-Regressive Integrated Moving Average Model）。ARIMA 模型并不是一个特定的模型，而是一类模型的总称。通常，我们用 p, d, q 值来确定一个特定的 ARIMA 模型，记作 $ARIMA(p, d, q)$ 。其中， p 代表自回归模型阶数， d 代表差分阶数， q 代表移动平均阶数。

ARIMA 模型的优点是数学原理简单，只需要内生变量（历史上的情况），不需要外生变量。但是缺点也很明显，本质上只能挖掘线性关系，不适用于非线性关系的时序数据。

13.3.2 重要概念

1. 平稳性

对于每一个统计学问题，我们都需要对其先做一些假设。比如最简单的线性回归，我们需要假设：①自变量是独立且随机的；②误差值服从均值为 0 的正态分布。同样，对于时间序列模型，最重要的假设是平稳性。从本质上讲，时间序列模型是通过历史规律预测未来的，即我们希望挖掘出的历史数据的一些性质在将来保持不变，这样对模型的预测才有意义。而平稳性就是用来刻画时间序列的统计性质关于时间平移的不变性的。如图 13-1 所示，上图代表趋势不平稳，下图代表趋势平稳。

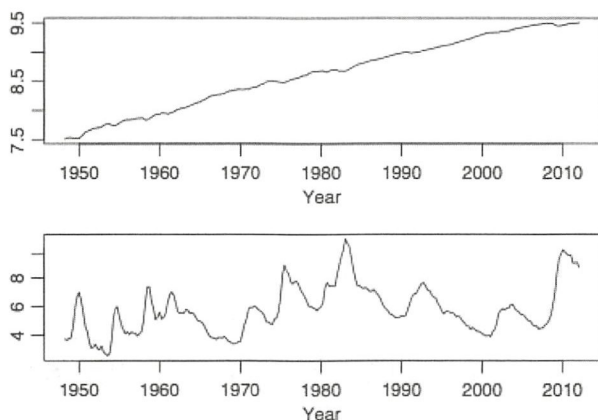


图13-1 时间序列是否平稳示意图

平稳性分为强平稳和弱平稳。强平稳是指时间序列 $\{Z_t\}$, 对于所有的 n 和 k , 满足 $Z_{t1}, Z_{t2}, \dots, Z_{tn}$ 和 $Z_{t1-k}, Z_{t2-k}, \dots, Z_{tn-k}$ 的联合分布相同。弱平稳是指时间序列 $\{Z_t\}$ 的均值是一个常数 μ , 不随时间变化, 并且协方差函数只与时间间隔相关。简单来理解, 强平稳是事实上的平稳(分布函数相同), 弱平稳是统计量在观测意义上的平稳(均值、方差)。通常, 时间序列模型仅基于弱平稳的假设。这是因为: ①在现实中, 很少有时间序列数据能够满足强平稳性; ②在统计学中, 当分布很复杂时, 很难写出其联合分布函数, 故而难以证明强平稳性。

2. 差分

差分是研究离散数学的一种方法, 类似于连续变量微分的概念。例如, 在微分计算中, $y = x^2$, 一阶导数 $y' = 2x$, 二阶导数 $y'' = 2$ 。举个实际生活中的例子, 原始函数 $f(x)$ 代表距离与时间的关系, 那么一阶导数则为速度, 二阶导数则为加速度。有了微分运算的概念之后, 我们将差分运算理解为适用于离散变量的微分运算即可。

在时间序列预测中, 差分是时间序列变量的本期值与上期值相减的运算。例如一阶差分为:

$$\Delta x_t = x_t - x_{t-1} \quad (13-9)$$

二阶差分为:

$$\Delta^2 x_t = \Delta x_t - \Delta x_{t-1} \quad (13-10)$$

那么, 在 ARIMA 模型中为什么需要差分? 在上文中我们已经知道, ARIMA 模型是基于平稳性这个假设前提的。但是在现实中, 大多数时间序列都是不平稳的, 例如随着时间的推移,

GDP 在一直增长。假设 GDP 随着时间 t 的增加而呈线性增长，如果对 GDP 进行一阶差分，那么就可以得到一个平稳的序列。

13.3.3 参数解释

ARIMA(p, d, q)由三个部分组成。

- AR(p): AR 是 Auto-Regressive 的缩写，表示自回归模型，含义是当前时间点的值等于过去若干个时间点的值的回归。因为不依赖其他的解释变量，只依赖自己过去的历史值，故称为自回归。如果依赖过去最近的 p 个历史值，则称阶数为 p ，记为 AR(p)模型。
- I(d): I 是 Integrated 的缩写，含义是模型对时间序列进行了差分。因为 ARIMA 要求平稳性，不平稳的序列需要通过一定的手段转化为平稳的序列，一般采用的手段是差分， d 表示差分的阶数。另外，还有一种特殊的差分是季节性差分 S ，即一些时间序列反映出一定的周期 T ，让 t 时刻的值减去 $t-T$ 时刻的值得到季节性差分序列。
- MA(q): MA 是 Moving Average 的缩写，表示移动平均模型，含义是当前时间点的值等于过去若干个时间点的预测误差的回归。预测误差=模型预测值-真实值。如果序列依赖过去最近的 q 个历史预测误差值，则称阶数为 q ，记为 MA(q)模型。

ARIMA 模型通用的表达式如下：

$$\hat{y}_t = \mu + \varphi_1 y_{t-1} + \cdots + \varphi_p y_{t-p} + \theta_1 e_{t-1} + \cdots + \theta_q e_{t-q} \quad (13-11)$$

其中， y_t 表示经过 d 阶差分后的变量。第一部分为常数 μ ，第二部分为 AR(p)自回归模型，第三部分为 MA(q)移动平均。

接下来，我们举例说明 ARIMA(p, d, q)模型中 p, d, q 取不同的值分别代表的意义以及数学表达式。

1. ARIMA(0,1,0)

ARIMA(0,1,0)又称作 Random Walk (随机行走)，如图 13-2 所示。每一时刻的 Y 值，只与上一时刻的 Y 值有关。也就是说，每一步的行走都是随机的。其计算公式如下：

$$\hat{Y}_t = \mu + Y_{t-1} \quad (13-12)$$



图13-2 随机行走示意图

2. ARIMA(1,0,0)

ARIMA(1,0,0)又称作 AR(1), 表示 t 期的 \hat{Y}_t 依赖 $t-1$ 期的 Y_{t-1} , 并且存在线性关系。其计算公式如下:

$$\hat{Y}_t = \mu + \varphi_1 Y_{t-1} \quad (13-13)$$

3. ARIMA(1,1,0)

ARIMA(1,1,0)表示一阶差分后 ($d=1$), t 期的差分变量 \hat{y}_t 依赖 $t-1$ 期的差分变量 y_{t-1} , 并且存在线性关系。其计算公式如下:

$$\hat{y}_t = \mu + \varphi_1 y_{t-1} \quad (13-14)$$

4. ARIMA(0,0,1)

ARIMA(0,0,1)又称作 MA(1), 表示 t 期的 \hat{Y}_t 值依赖 $t-1$ 期的预估误差值 e_{t-1} , 并且存在线性关系。其计算公式如下:

$$\hat{Y}_t = \mu + \theta_1 e_{t-1} \quad (13-15)$$

5. ARIMA(0,1,1)

ARIMA(0,1,1)表示一阶差分后 ($d=1$), t 期的差分变量 \hat{y}_t 依赖 $t-1$ 期的预估误差值 e_{t-1} , 并且存在线性关系。其计算公式如下:

$$\hat{y}_t = \mu + \theta_1 e_{t-1} \quad (13-16)$$

13.3.4 建模步骤

本节中，我们将为大家介绍通用的 ARIMA 模型建模步骤（如图 13-3 所示）。首先，对时间序列（简称时序）数据进行平稳性检验，若不通过，则采取取对数、差分等方法使时序数据具有弱平稳性。通过平稳性检验后，进行白噪声检验，当序列不是白噪声序列时，即可选择合适的 ARIMA 模型进行拟合。模型拟合后，对误差值进行白噪声检验，通过后分析结束，采用拟合出的 ARIMA 模型对时序数据进行预测。

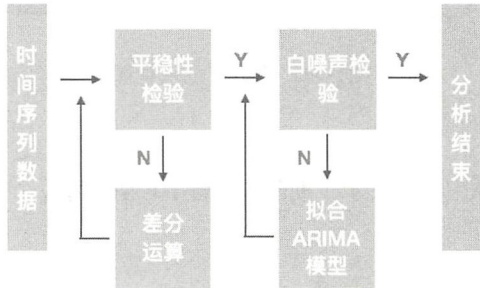


图13-3 ARIMA模型建模步骤

1. 数据平稳性检验

上文提到，ARIMA 模型是基于时序数据平稳性这个假设前提的，因此在进行模型拟合前，需要对时序数据进行平稳性检验。通常，我们先画出时序散点图或折线图，通过肉眼判定是否平稳。若是明显不平稳的序列，则可进行取对数、差分等预处理。在肉眼识别后，可对时序数据进行 ADF 单位根检验，从统计学上判定是否平稳。

ADF 单位根检验的原假设是存在单位根，因此如果得到的统计量显著小于 3 个置信度（1%、5%、10%）的临界统计值，则说明拒绝原假设，即该时序数据不存在单位根，呈现平稳性；否则，接受原假设，即该时序数据存在单位根，是非平稳的序列，需要进一步进行数据处理，以得到平稳的序列。

2. 白噪声检验

白噪声序列，是指对于一个不相关的随机变量 $\{X_t\}$ ，均值为 0，方差为常数。之所以称为白噪声，是因为它和白光的特性类似，白光的光谱在各个频率上有相同的强度，白噪声的谱密度在各个频率上的值相同。

根据定义，白噪声序列需要满足的数学条件为：

- $E(X_t) = 0$ ，均值为 0。

- $\text{Var}(X_t) = \mu$, 方差为常数。
- $\text{Cov}(X_t, X_s) = 0 \ (t \neq s)$, 协方差为 0。

在时间序列中序列值彼此之间若没有任何相关性, 则是不能建立模型进行分析的。因为一个纯随机序列意味着 t 时刻的值完全是随机的, 并不依赖上一时刻的变量值, 这显然不符合 ARIMA 模型的基本思想。

那么, 如何进行白噪声检验呢? 我们采用的方法是 Ljung-Box Test, 简称 LB 检验。LB 检验基于一系列滞后阶数, 判断序列是否存在总体的相关性或者随机性。其原假设和备择假设为:

H_0 : 数据不存在相关性, 是相互独立的。

H_1 : 数据存在相关性, 不是相互独立的。

构造的统计量, 称作 Q 统计量:

$$Q = n(n+2) \sum_{k=1}^h \frac{\hat{p}_k^2}{n-k} \quad (13-17)$$

其中, n 是样本数量, \hat{p}_k^2 是样本 k 阶滞后的相关系数, 该统计量服从自由度为 h 的卡方分布。给定显著性水平 α , 若 $p \leq \alpha$, 则拒绝原假设, 即原序列不是白噪声序列, 存在相关性, 可进一步选定合适的 ARIMA 模型进行拟合。

3. ARIMA模型选型

我们已经知道, ARIMA 模型不是一个特定的模型, 而是一系列模型的总称, 它由三个参数构成, 计作 $\text{ARIMA}(p, d, q)$ 。对于不同的时序数据, 我们需要选定不同的 p, d, q 值进行模型拟合。

通常采用自相关函数 (ACF)、偏自相关函数 (PACF) 来判别 ARIMA 模型的 q 和 p 值。ACF 描述时间序列观测值与其过去的观测值之间的线性相关性, PACF 描述在给定中间观测值的条件下时间序列观测值与其过去的观测值之间的线性相关性。平稳的随机时间序列模型特征系数如表 13-1 所示。

表 13-1 平稳的随机时间序列模型特征系数

平稳的随机时间序列模型	ACF	PACF
$\text{AR}(p)$	拖尾	p 阶截尾
$\text{MA}(q)$	q 阶截尾	拖尾
$\text{ARMA}(p, q)$	拖尾	拖尾

p 由显著不为 0 的 PACF 的数目决定, q 由显著不为 0 的 ACF 的数目决定。这里的拖尾是指以指数率单调或振荡衰减 (如图 13-4 左图所示), 截尾是指从某个数目开始非常小, 且趋近零 (理想情况是 PACF 在 p 阶之后为 0, 如图 13-4 右图所示)。在平稳的时间序列中, 应用 ACF 和 PACF 在初步判断 ARIMA 模型的阶数 p 和 q 的基础上, 通过最小信息准则 AIC 进行定阶。

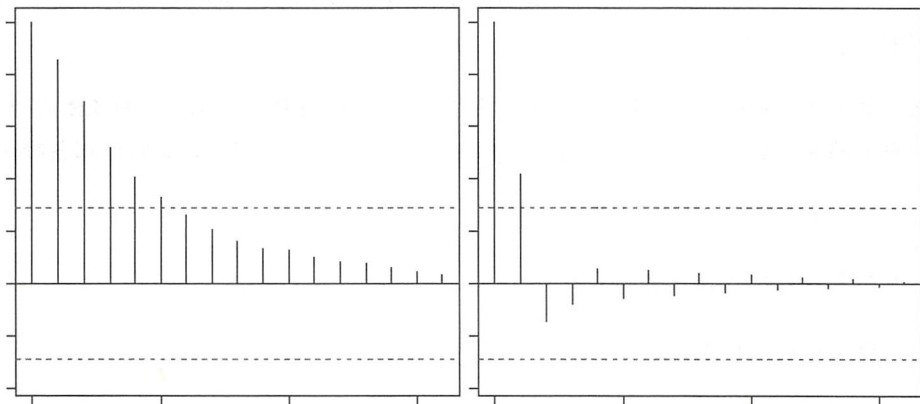


图13-4 拖尾和截尾示意图

4. ARIMA模型验证

模型拟合之后, 应对通过模型取得的估计结果进行检验与诊断, 以验证所选用的模型是否合适。这一过程主要检验所拟合的时间序列模型是否客观、合理。针对模型的合理性检验, 通常从两个方面进行判断。

- 要验证所拟合的时间序列模型的参数估计值是否有显著性。
- 要验证所拟合的时间序列模型的残差序列是否是白噪声序列, 即进行残差序列的独立性检验。

若这两项验证通过, 则认为该模型是合理的; 否则, 应重新选取有效的模型, 然后应用该模型进行预测。

13.3.5 ARIMA模型案例

本节中, 我们以一周的微博广告曝光时序数据 (间隔为 1 分钟) 为例, 拟合 ARIMA 模型 (示例代码用 Python 编写)。

1. 数据预处理

此部分包含数据读取、格式转换、构建时间序列数据并进行平滑。利用加权移动平均法平滑后的时序数据趋势如图 13-5 所示。

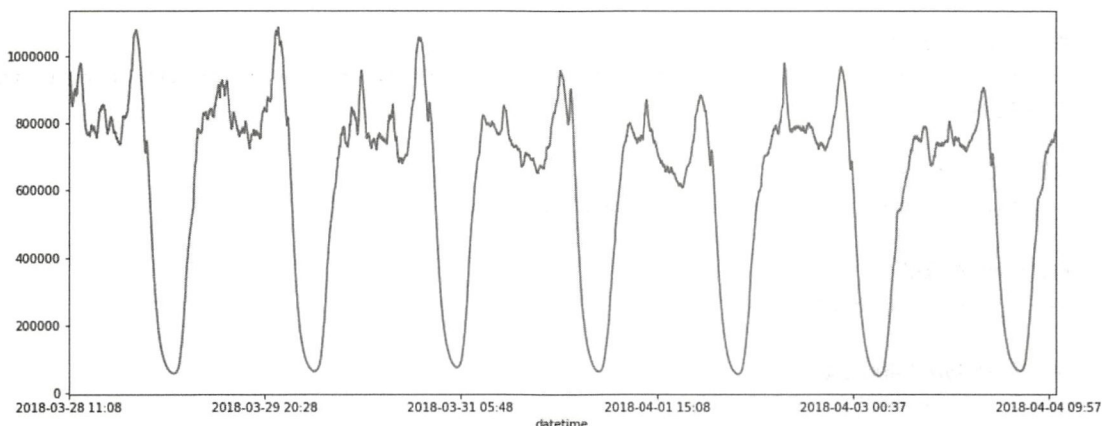


图13-5 利用加权移动平均法平滑后的时序数据趋势图

```
##时间序列分析 ARIMA 模型
# 加载分析所需要的包
from urllib import request
import json
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.stats.diagnostic import acorr_ljungbox
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.tsa.arima_model import ARIMA
import statsmodels.api as sm
import numpy as np
import time

# 读取数据
url = "这里是数据 url"
wp = request.urlopen(url)
content = wp.read().decode()
array = json.loads(content)
data = array[0]['datapoints']

# 将数据转换为 data frame
df = pd.DataFrame(data)
```



```

df.columns = ['imp_cnt', 'time']
df_dropna = df.dropna(axis=0, how='any')

# 按时间排序
df_sort = df_dropna.sort_values(['time'])

# 将时间戳转换为时间
df_sort['datetime'] = [time.strftime("%Y-%m-%d %H:%M", time.localtime(i)) for i in
list(df_sort.time)]

# 设置时间为 index
df_sort.set_index(["datetime"], inplace=True)

# 构建时间序列数据
ts = df_sort['imp_cnt']

# 通过滑动平均法平滑时序数据
ts_ewma = pd.ewma(ts, span = 60)
plt.figure(figsize = (15,6))
ts_ewma.plot()
plt.show()

```

2. 拟合ARIMA模型

此部分包含差分、平稳性检验（ADF）、白噪声检验、ARIMA 模型定阶、模型拟合。其中，根据图 13-6 所示的 ACF 和 PACF 情况可以看出，ACF 呈现拖尾，PACF 在 6 阶表现为截尾。因此选择 ARIMA(6,1,0)进行拟合。我们用一周的数据进行测试，如图 13-7 所示，前 5 天（5 月 23 日~5 月 27 日）的数据为原始值，后 2 天（5 月 28 日~5 月 29 日）为拟合后的预测值。可以看出，预测的数据符合整体趋势情况，拟合效果较好。

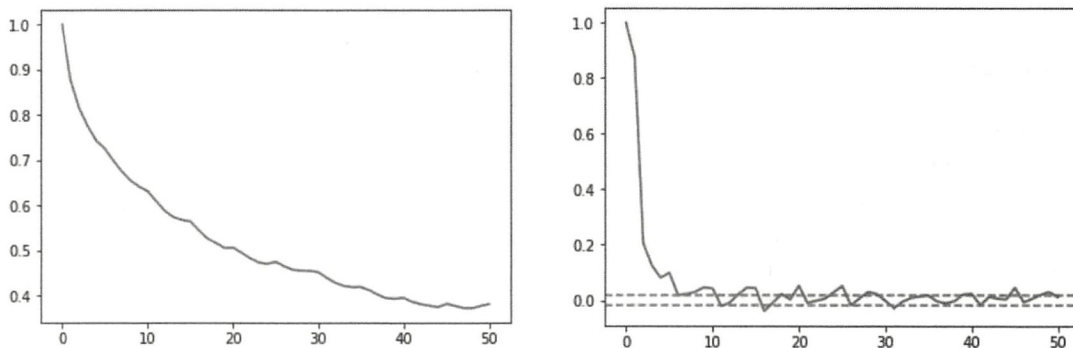


图13-6 ACF（左）和PACF（右）分布图

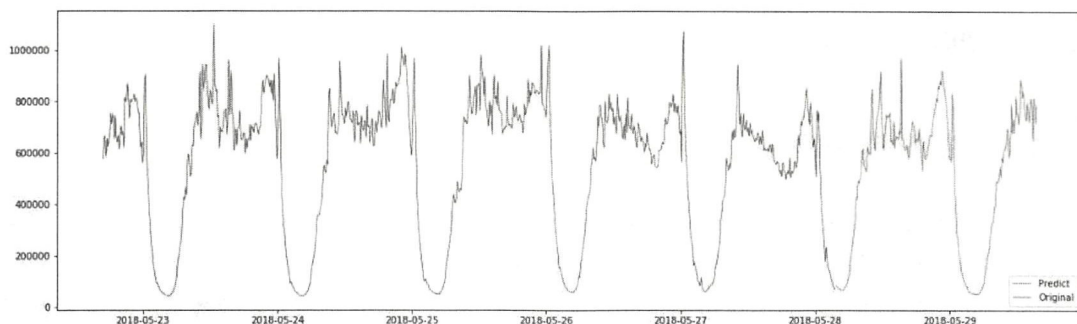


图13-7 预测值和原始值时间序列趋势图（前5天为原始值，后2天为预测值）

```
##一阶差分
ts_diff_1 = ts_ewma.diff(1).dropna(axis=0, how='any')
# ADF 检验平稳性
adfuller(ts_diff_1, autolag='AIC')
# 白噪声检验
acorr_ljungbox(ts_diff_1, 1)
# ACF、PACF
lag_acf = acf(ts_diff_1, nlags=50)
lag_pacf = pacf(ts_diff_1, nlags=50)
plt.plot(lag_acf)
plt.show()
plt.plot(lag_pacf)
plt.axhline(y=-1.96/np.sqrt(len(ts_diff_1)), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(ts_diff_1)), linestyle='--', color='gray')
plt.show()
# 拟合模型
model = ARIMA(ts_diff_1, order=(6, 0, 0))
ts_predict = model.fit().predict()
rmse = np.sqrt(sum((ts_predict-ts_diff_1)**2)/ts_diff_1.size)
# 数据可视化
plt.figure(facecolor='white', figsize = (20,6))
plt.plot(ts_predict[:1440], lw = 0.5, color='blue', label='Predict')
plt.plot(ts_diff_1[:1440], lw = 0.5, color='red', label='Original')
plt.legend(loc='lower right')
plt.ylim((-11000, 11000))
plt.show()
```

13.4 神经网络模型

13.4.1 卷积神经网络

卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，CNN 的卷积层的神经元只与前一层的部分神经元节点相连，即它的神经元间的连接是非全连接的，从而减少了需要训练的参数数量，对于大型图像处理有出色的表现。

CNN 属于神经网络的一种，图 13-8 展示了神经网络结构示意图，其中 Layer 1 为输入单元，Layer 2 和 Layer 3 为隐藏层，Layer 4 为输出单元。

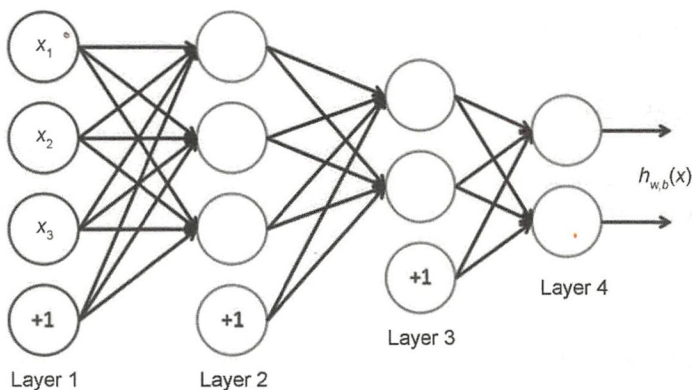


图13-8 具有两个隐藏层的神经网络结构示意图

CNN 可以包含很多层，其中核心单元是卷积层和池化层。

1. 输入层（Input Layer）

输入层主要对原始数据进行预处理。在图像处理中，输入层的预处理通常包括去均值、PCA 降维、对数据各个特征轴上的幅度归一化（白化）等。

2. 卷积层（Convolutional Layer）

卷积层是 CNN 的核心单元，它使用卷积核进行特征提取和特征映射。在这一层中，每个神经单元都被看作一个滤波器（Filter）。卷积计算过程是按照一定的步长（Stride），通过一个窗口区域（Receptive Field）进行滑动扫描产生下一层神经元的值，其中窗口区域权重矩阵 w 被称为卷积核，即卷积神经网络的参数。为了有一个偏移项，卷积核可附带一个偏移量 b ，它们的

初值可以随机生成, 并且可以通过训练进行变化。这样, 通过公式 (13-18) 可以计算下一层神经元的值。

$$b + \sum_{i=0}^s \sum_{j=0}^s w_{ij} x_{ij} \quad (13-18)$$

其中, w 为卷积核, x 为上一层神经元的值, b 为偏移量, s 为步长。

图 13-9 展示了一个 7×7 的输入单元经过步长为 3、窗口区域为 3×3 的卷积计算过程示意图。

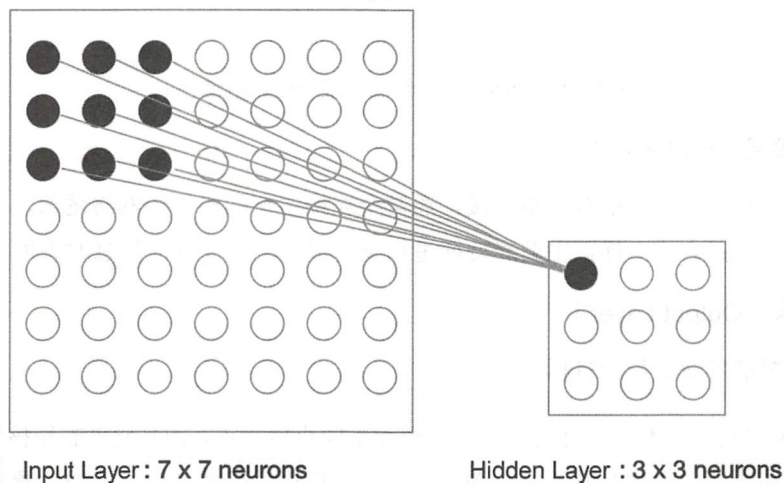


图13-9 卷积计算过程示意图 (其中步长为3, 窗口区域为 3×3)

3. 激励层 (ReLU Layer)

激励层主要对卷积层的输出进行一个非线性映射, 因为卷积层的计算还是一种线性计算。CNN 采用的激励函数一般为 ReLU (Rectified Linear Unit, 修正线性单元) 函数, 它的特点是收敛快, 求梯度简单。

$$f(x) = \max(x, 0) \quad (13-19)$$

卷积层和激励层通常合并在一起称为“卷积层”。

4. 池化层 (Pooling Layer)

池化层主要用于特征降维, 进行下采样, 对特征图稀疏处理, 减少数据运算量, 通过压缩数据和参数量, 也能够减小过拟合。

图 13-10 展示的是池化层将 4×4 维降低到 2×2 维的过程,按照取最大值池化(Max Pooling)的方法实现,通常也可以采用取平均值池化(Average Pooling)的方法。

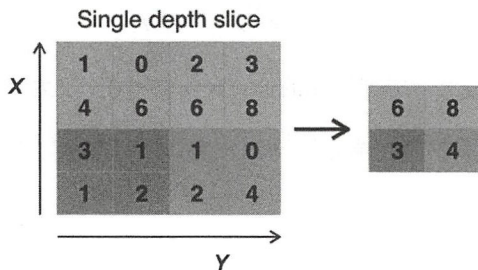


图13-10 CNN池化层示例(采用最大值池化)

5. 全连接层 (FC Layer)

两层之间所有的神经元都有权重连接,通常全连接层在卷积神经网络尾部,即与传统的神神经网络神经元的连接方式一样,主要对特征进行重新拟合,减少特征信息的丢失。

6. 输出层 (Output Layer)

输出层主要准备做好最后目标结果的输出。

对于 CNN 来说,并不是所有的上下层神经元都能直接相连,而是将“卷积核”作为中介。同一个卷积核在所有图像内是共享的,图像通过卷积操作后仍然保留原先的位置关系。对于图像,如果没有卷积操作,学习的参数量是灾难级的。CNN 之所以用于图像识别,正是由于 CNN 模型限制了参数的个数并挖掘了局部结构的这个特点。此外,在卷积层后的池化层(一般采用最大值池化)可以进一步提取特征,降低维度。

13.4.2 循环神经网络

在现实场景中,全连接的 DNN (Deep Neural Networks, 深度神经网络)容易出现参数数量的膨胀,不仅容易过拟合,而且极易陷入局部最优。同时,存在的另一个问题是无法对时间序列上的变化进行建模。在 CNN 中,当前层神经网络只跟上一层神经网络有连接,降低了 DNN 的维度爆炸和造成梯度消失的风险。然而,DNN 和 CNN 都没有考虑样本出现的时间顺序,这样的时间顺序对于自然语言处理、语音识别、手写体识别等应用非常重要。为了适应这种需求,就出现了另一种神经网络结构——循环神经网络(RNN)。在 RNN 中,神经元的输出可以在下一个时间戳直接作用到自身。RNN 可以被看成一个在时间上传递的神经网络,它的深度是时间

的长度,如图 13-11 所示。

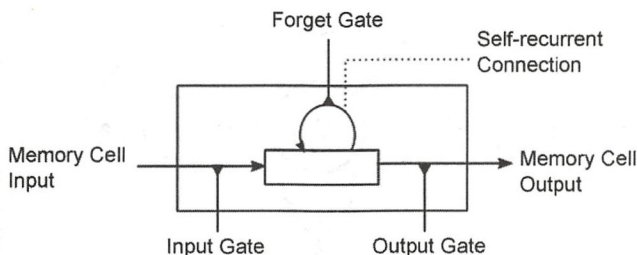


图13-11 RNN示意图 (图片来源: *Deep Learning Tutorials*)

理论上,虽然 RNN 可以解决序列数据的训练问题,但是它也像 DNN 一样存在梯度消失的问题,当序列很长时问题尤其严重。因此,RNN 模型一般不能直接用于应用领域。在语音识别、手写体识别以及机器翻译等 NLP 领域应用比较广泛的是基于 RNN 模型的一个特例 LSTM,下一节我们就来讨论 LSTM 模型。

13.4.3 长短期记忆网络

RNN 还有许多变形,例如双向 RNN (Bidirectional RNN) 等。然而,RNN 在处理长期依赖(在时间序列上距离较远的节点)时会遇到巨大的困难,因为在计算距离较远的节点之间的联系时会涉及雅可比矩阵的多次相乘,这会带来梯度消失(经常发生)或者梯度膨胀(较少发生)的问题,这样的现象被许多学者观察到并独立研究。为了解决该问题,研究人员提出了许多解决办法,例如 ESN (Echo State Network)、增加泄漏单元 (Leaky Unit) 等。其中应用最成功、最广泛的就是门限 RNN (Gated RNN),而 LSTM 就是门限 RNN 中最著名的一种。

LSTM^[1] (Long Short-Term Memory) 即长短期记忆网络,由 Hochreiter & Schmidhuber (1997) 提出,并被 Alex Graves 进行了改良和推广。LSTM 区别于 RNN 的地方,主要就在于它在算法中加入了一个判断信息是否有用的“处理器”,这个处理器作用的结构被称为记忆单元 (Memory Cell),如图 13-12 所示。在一个 Cell 中被放置了三扇门,分别叫作输入门、遗忘门和输出门。当一个信息进入 LSTM 中后,可以根据规则来判断其是否有用,只有符合算法认证的信息才会留下,不符合的信息则通过遗忘门被遗忘。

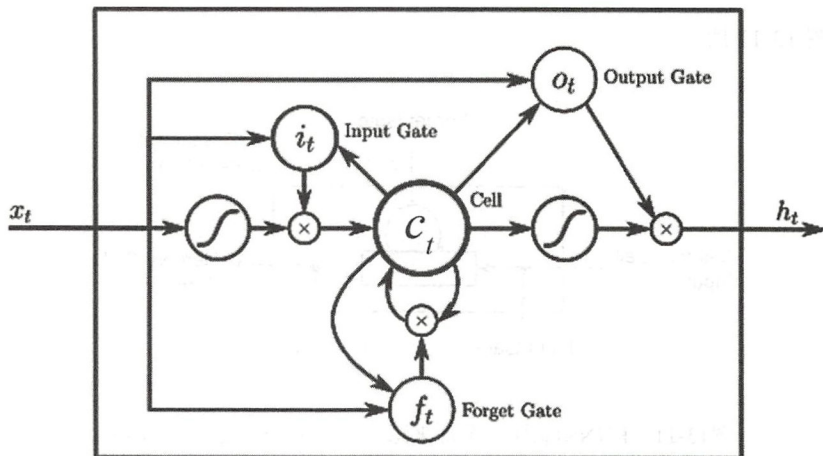


图13-12 LSTM示意图

LSTM 包含 4 个非常关键的元素，即输入门（Input Gate）、输出门（Output Gate）、遗忘门（Forget Gate）和记忆单元。在 LSTM 模型中，各个部分的计算公式如下：

(1) 输入门

$$i_t = \delta(W_i x_t + U_i h_{t-1} + b_i) \quad (13-20)$$

(2) 输出门

$$o_t = \delta(W_o x_t + U_o h_{t-1} + b_o) \quad (13-21)$$

(3) 遗忘门

$$f_t = \delta(W_f x_t + U_f h_{t-1} + b_f) \quad (13-22)$$

(4) 记忆单元

$$C'_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \quad (13-23)$$

$$C_t = f_t * C_{t-1} + i_t * C'_t \quad (13-24)$$

最后得到 LSTM 单元的输出向量为：

$$h_t = o_t * \tanh(C_t) \quad (13-25)$$

其中，初始化时 $C_0 = 0$, $h_0 = 0$, LSTM 的输入单元用 x_t 表示, LSTM 的输出单元用 h_t 表示,

用 δ 表示 sigmoid 函数:

$$\delta(x) = \frac{1}{1+e^{-x}} \quad (13-26)$$

tanh 函数表达式为:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (13-27)$$

目前已经证明, LSTM 是解决长序依赖问题的有效技术, 并且这种技术的普适性非常高, 导致带来的可能性变化非常多。研究者们根据 LSTM 纷纷提出了自己的变体版本, 在实际应用中, 这些 LSTM 的变体被广泛应用到各个领域, 读者可以查阅相关文档进行了解 (推荐阅读 *Understanding LSTM Networks*^[2], 以及示例 *LSTM implementation explained*^[3])。

13.4.4 应用说明

使用传统方法 (ARIMA、Holter-Winter) 对实际数据的预测效果往往不能令人满意, 比如 ARIMA 算法的一个技术难点就是时间序列的平稳化, 平稳化的时间序列对于预测结果的好坏起着至关重要的作用。另一个问题是对数据进行滑动平均操作, 容易造成波动锯齿明显, 直接用于报警会造成误报干扰。

相对于现有的预测算法, LSTM 对监控数据的预测准确性较高, 它充分考虑到时间记忆, 符合时序数据的特点, 在智能运维领域尤其是在趋势预测、动态阈值等方面具有一定的研究价值。本书后面章节还会再次讨论 LSTM 在微博广告监控系统中的详细应用和实践。

13.5 本章小结

本章重点讨论了趋势预测尤其是时序数据趋势预测的相关算法和模型, 并重点介绍了移动平均法和指数平滑法在趋势预测领域的技术方案。同时还介绍了 ARIMA 模型的建模步骤和模型应用案例。最后介绍了神经网络相关技术尤其是 LSTM 模型, 它能够被很好地应用在时序数据预测方面。

13.6 参考文献

- [1] Long Short-Term Memory. Neural Computation. Volume 9, Issue 8. 1735-1780, 1997
- [2] Understanding LSTM Networks. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- [3] LSTM implementation explained. <http://apaszke.github.io/lstm-explained.html>

第 4 篇

智能运维架构实践：技术案例详解

人工智能（AI，Artificial Intelligence）不断爆发热潮，与基础设施的进步和科技的更新是分不开的，从 20 世纪 70 年代个人计算机的兴起到 2010 年 GPU、异构计算等硬件设施的发展，都为人工智能的复兴奠定了基础。同时，互联网及移动互联网的发展也带来了一系列数据能力，使人工智能得以提高。

随着大数据的存储计算能力不断升级，开源项目日渐增多，以及这些开源平台社区的不断成熟，让更多的企业能够基于开源技术构建自己的大数据平台以及机器学习平台，极大地降低了开发成本。

对于智能运维体系而言，监控平台是其中一个非常典型的和核心的部分，它涉及大规模数据的收集、存储、数据建模、分析处理以及可视化等。同时，结合机器学习框架，它也为模型的呈现提供了基础框架。开源的 ELK（Elasticsearch, Logstash, Kibana）组合套件，能够快速构建基于大规模数据的分布式监控平台，再结合 Zabbix 可以快速集成报警系统。这些开源框架在微博中得到了极大的应用，经过了长时间的实践检验。

微博广告的运维系统更偏于业务，与业务结合得更紧密；而微博平台的运维系统更偏于通用型，与具体业务的耦合性较低。这两个运维平台非常具有代表性，因此本篇将主要通过微博广告和微博平台两个具体业务场景下的具体案例进行阐述，系统、全面地剖析智能监控系统架构及系统设计原理。本篇主要分为以下几章：

- 第 14 章 快速构建日志监控系统
- 第 15 章 微博广告智能监控系统
- 第 16 章 微博平台通用监控系统

第 14 章

快速构建日志监控系统

Elasticsearch、Logstash 和 Kibana 三个开源框架的套装组合被称作 ELK，它们分别负责数据检索、数据收集和数据可视化，开发者基于 ELK 可以非常容易搭建起支撑 PB 级别日志量的大数据分析平台和监控系统。本章内容将重点关注 Elasticsearch、Kibana 的基本原理，以及如何基于 ELK 搭建分布式日志监控系统。

14.1 Elasticsearch 分布式搜索引擎

Elasticsearch^[1]是一个分布式的、可扩展的、实时的搜索与数据分析引擎。Elasticsearch 是基于 Apache Lucene 构建而成的，而 Apache Lucene 是一个成熟的、高性能的、功能强大的全文检索和搜索库，因此其无论在性能还是功能方面，都继承了 Apache Lucene 的优势。同时，它隐藏了 Lucene 操作的复杂性，通过 REST API 让索引和检索都变得简单。Elasticsearch 还支持数据分片，每个分片底层都是一个 Lucene 索引，这让 Elasticsearch 拥有了更强的扩展性和更好的搜索性能。

14.1.1 基本概念

1. 集群

集群由一个或多个节点组成，集群保存着所有的数据，并在所有节点上提供索引和搜索功能。集群需要有一个唯一的名称标识，默认的名称为“elasticsearch”。节点通过这个唯一的名称标识来加入对应的集群。

2. 节点

一个单独的 Elasticsearch 实例称为一个节点。在大多数情况下，建议在一台服务器上运行

一个节点。在 Elasticsearch 集群中节点拥有以下几个角色。

(1) 主节点 (Master Node)

主节点负责轻量级的集群范围的操作，例如创建或删除索引，跟踪哪些节点是集群的一部分，以及决定将哪些分片分配给哪些节点。具有稳定的主节点对于群集健康来说是重要的。任何合格的节点（默认所有节点）都可以通过主选举过程选择成为主节点。主节点必须能够访问数据/目录（就像数据节点一样），因为这是集群状态在节点重新启动时持久化的地方。索引和搜索数据是 CPU、内存和 I/O 密集型工作，可能会对节点的资源造成压力。为了确保主节点稳定且不受压力，在较大的集群中建议分开主节点和数据节点。虽然主节点也可以作为协调节点，并将搜索和索引请求从客户端路由到数据节点，但建议不要将它们共用在一个节点上，还是应该分开主节点和协调节点。为了增强集群的稳定性，主节点应该做尽可能少的工作。同时，为了防止脑裂行为，出现数据丢失，必须通过设置 `discovery.zen.minimum_master_nodes` 来控制一个集群形成前，集群内拥有最少的主节点数量。`discovery.zen.minimum_master_nodes` 默认值为 1，可以通过公式 $(\text{master_eligible_nodes}/2)+1$ 来决定该设置的值，其中 `master_eligible_nodes` 代表主节点的数量。

(2) 协调节点 (Coordinate Node)

如果主节点因为高负荷失去了处理任务、保存数据和预处理文档的能力，那么可以通过增加协调节点来解放主节点的一部分工作。协调节点可以接收搜索请求或者批量索引请求，并将请求分散转发到保存数据的数据节点，每个数据节点在本地执行请求并将其结果返回给协调节点。在收集阶段，协调节点将每个数据节点的结果减少为单个全局结果集。每个节点都可以是协调节点。我们可以通过在一个节点的配置文件中将 `node.master`、`node.data` 和 `node.ingest` 设置为 `false` 来让该节点成为协调节点，协调节点不能被禁用。所以，协调节点需要拥有高性能的 CPU 和内存，以便处理聚合的数据。协调节点不应该设置得过多，否则会增加集群的负担，因为主节点必须等待所有节点的集群状态更新确认。

(3) 数据节点 (Data Node)

数据节点包含建立索引的文档分片。数据节点用于处理数据相关操作，如 CRUD、搜索和聚合，这些操作是 I/O、内存和 CPU 密集型操作。所以，应该密切监视数据节点的资源使用情况，如果数据节点过载，则意味着需要添加更多的数据节点。

(4) Ingest 节点

Ingest 节点可以执行预处理管道，它由一个或多个 Ingest 处理器组成。

（5）Tribe 节点

Tribe 节点用于连接多个 Elasticsearch 集群。

除上面提到的几个节点外，现在还有 X-pack 节点和机器学习节点。

3. 索引

Elasticsearch 把数据存放到一个或者多个索引中。如果用关系数据库模型来对比，一个索引就相当于一个库。索引存放和读取的基本单元是文档。

4. 分片

Elasticsearch 是通过分片将一份数据分布到多个节点上的。因此，一个索引可能包含多个分片，而每个分片都是一个 Lucene 索引。系统默认一个索引有 5 个分片，虽然系统会为我们自动创建分片，但是一个索引的主分片创建完成后就不能再被修改了，所以需要提前计算好一个索引大约会存储多少数据，需要几个分片，并在索引生成前配置好该索引分片的个数。我们可以在索引模板中设置主分片数量：

```
{
  "template": "logstash-*",
  "settings": {
    "number_of_shards": 10,
    "number_of_replicas": 1
  }
}
```

这里设置了以 logstash 开头的索引主分片数量为 10 个，每个主分片的副本分片为 1 个。

分片的数量并不是越多越好，越多的分片虽然能带来数据索引上的高效，但同时也会增加数据检索的成本。因此需要衡量数据的索引和检索，为每个索引指定一个合适的分片数。同时，应该注意单个分片的大小最好不超过 10GB。

5. 副本

副本是实现集群高可用的关键。通过数据分片将一个索引中的数据分成多份存储在不同的分片中。但是，当遇上机器宕机、系统异常、某个数据节点的 Elasticsearch 服务不可用时，该节点上的数据也将不能被访问。因此需要对每个分片复制一份或者多份，使它可以像主分片一样接收处理用户的请求。这样，当一个节点不可用时，依然可以从该分片的副本节点上请求到数据，从而保障了数据的高可用性和安全性。

相对于主分片数量在索引生成后不可调整，副本分片是可随时添加或者删除的。可以通过下面的设置在需要的时候动态调整副本分片的数量：

```
PUT logstash/_settings
{
  "number_of_replicas": 2
}
```

6. Segment

每个分片都包含多个 Segment，每个 Segment 都是一个倒排索引。

数据在被写入 Elasticsearch 时，会先被写到内存 Buffer 中，此时数据还不能被查询。为了安全起见，从数据被写到内存 Buffer 开始，Elasticsearch 会记录一个 translog 日志，在确认数据写到磁盘之前，如果出现异常，就可以通过 translog 恢复记录。

当数据被写到内存 Buffer 后，经过指定的时间间隔，Elasticsearch 将会把内存中的数据刷（flush）到文件系统缓存中的 Segment 中，此时数据就可以被查询了。默认间隔时间为 1 秒，可以通过调整 refresh_interval 参数来修改刷新间隔，如果对数据实时性要求不高，则可以调大该刷新时间。但是此时由于数据还没被写到磁盘上，会有丢失数据的风险，所以 Elasticsearch 还会执行 flush 操作，将文件系统缓存中的 Segment 真正写入磁盘文件中，并清空 translog 数据。关于 Elasticsearch 执行 flush 的时间，默认为 5 秒，或者当 translog 文件大小超过 512MB 时会执行一次刷新，也可以通过下面两个参数来修改这个刷新时间。

- index.translog.flush_threshold_size: 当 translog 文件大小超过指定值时，将文件系统缓存中的 Segment flush 到磁盘上。默认为 512MB。
- index.translog.sync_interval: 指定将文件系统缓存中的 Segment flush 到磁盘上的时间间隔，默认为 5s，不允许低于 100ms。

Elasticsearch 会定期将分片中的 Segment 进行合并，生成一个大的 Segment，并删除小的 Segment。

7. 文档

文档是存储数据的实体。一个文档由一个或者多个字段组成。在同一个索引中，文档与文档之间也可以由不同的字段组成。数据在被存储到文档之前，通常会经过分析筛选，用户会将日志重构成 key:value 的结构化形式，然后再存储到索引里面。一个文档不只包含它的数据部分，还包含文档的元数据。

- `_index`: 文档所在的索引名称。
- `_type`: 文档的类型。在 Elasticsearch 6.0 之前, 在同一个索引中可以包含多种文档类型; 从 Elasticsearch 6.0 开始, 系统默认都是 `doc` 类型。
- `_id`: 文档的唯一标识。

14.1.2 分布式文档存储与读取

上面介绍了 Elasticsearch 集群的基础概念, 那么一条数据究竟该如何被索引到集群中, 又如何从集群中检索呢?

1. 文档索引

如图 14-1 所示, 文档索引的步骤如下:

- A. 客户端发送一个索引请求给 Node.1 节点。
- B. Node.1 节点默认使用 `_id` 来决定该文档存储在 P2 上, Node.1 将请求转发给 P2 主分片所在的节点 Node.2。
- C. Node.2 节点在主分片上处理请求, 如果成功, 则同时将请求转发给两个副本分片所在的节点 Node.1 和 Node.3。
- D. 一旦两个副本分片反馈写入成功, 主分片所在的 Node.2 节点就向协调节点 (这里是 Node.1) 返回成功响应信息。
- E. 协调节点 (这里是 Node.1) 再向客户端返回成功响应信息。

所以, 当客户端接收到成功响应信息时, 文档已经在主分片和副本分片上被成功处理了。

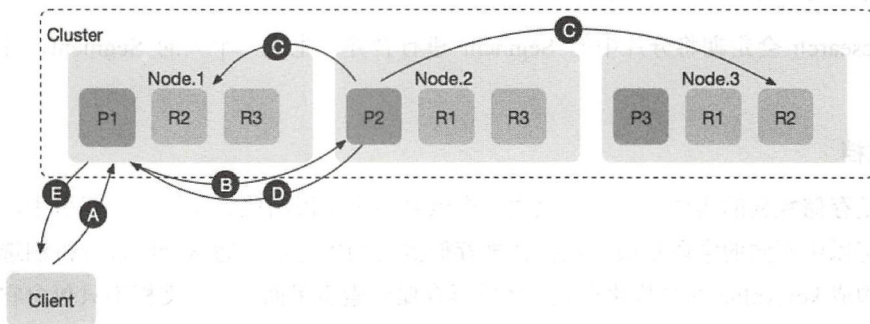


图14-1 文档索引

备注：为了简化，图中 P 代表数据主分片，R 代表数据副本，比如 Node.1 节点上的 P1 对应的副本为 R1，分别在 Node.2 和 Node.3 节点上存储，后续章节将沿用此命名方法。

2. 文档更新

如图 14-2 所示，文档更新的步骤如下：

- A. 客户端发送一个请求到 Node.1 节点。
- B. Node.1 节点根据路由的结果请求转发到主分片所在的 Node.3 节点上。
- C. Node.3 节点根据 `_id` 从主分片上检索文档，修改 `_source` 字段中的内容，然后尝试重新索引该文档。如果此时该文档被另一个请求所修改，那么它将重复上面的操作，直到成功或者超过 `retry_on_conflict` 参数设置后放弃。如果文档索引成功，Node.3 节点将把新版本的文档转发到 Node.1 和 Node.2 节点上，在两个节点的副本分片上重新建立索引。
- D. Node.1 和 Node.2 节点在成功建立索引后，将向 Node.3 节点返回成功的信息。
- E. Node.3 节点将返回成功的信息给 Node.1 节点。
- F. Node.1 节点将返回成功的信息给客户端。

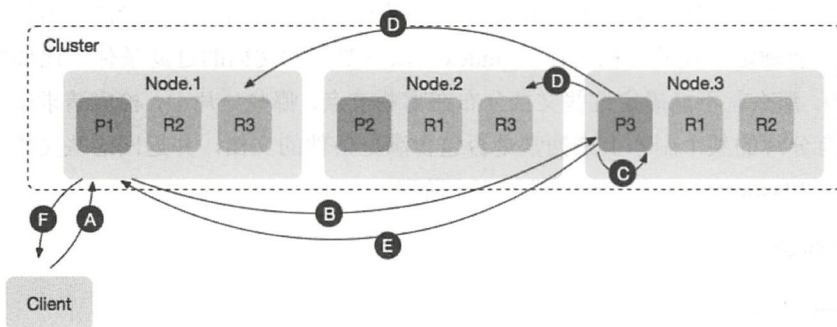


图14-2 文档更新

3. 读取文档

如图 14-3 所示，读取文档的步骤如下：

- A. 客户端发送一个请求到 Node.1 节点。
- B. Node.1 节点通过文档 `_id` 计算出该文档属于 P3，Node.1 节点发现 P3 的副本在所有三

个节点上，Node.1 节点随机将请求转发到一个节点 Node.2 上。

- C. Node.2 节点返回请求给 Node.1 节点。
- D. Node.1 节点将请求返回给客户端。

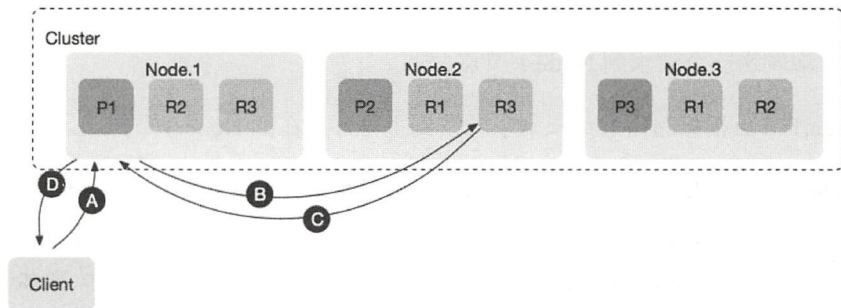


图14-3 读取文档

对于每次的请求，协调节点都通过轮询的方法遍历每个副本，以达到均衡的目的。

14.1.3 分布式文档检索

在上面的读取文档操作环节中，我们明确知道一个文档的 `_index`、`_id`，可以根据这些字段的组合来确定一个文档处于集群中的哪个节点、哪个分片上。而这里讨论的检索则是更复杂的操作。可能只能确定要找的文档所在的 `_index`，以及要找的文档的过滤条件，比如所有状态字段 `status=200`。那么就不能确定这些文档分布在哪些节点、哪些分片上，检索请求就会被协调节点转发到所有分片的某个副本上来搜索是否包含满足条件的文档，并返回这些文档。而这样一个操作流程在 Elasticsearch 内部被分成了两个阶段。

1. 查询阶段

如图 14-4 所示，查询阶段的步骤如下：

- A. 客户端发送请求到 Node.1 节点。
- B. Node.1 节点将请求转发到所有分片的主分片或者一个副本分片上。
- C. 每个分片都根据查询条件在本地执行查询请求，并将查询结果中每个文档的 `id` 和排序值返回给协调节点，该协调节点就是 Node.1。Node.1 节点合并每个分片返回的结果，在本地全局排序，生成一个总的排序列表。

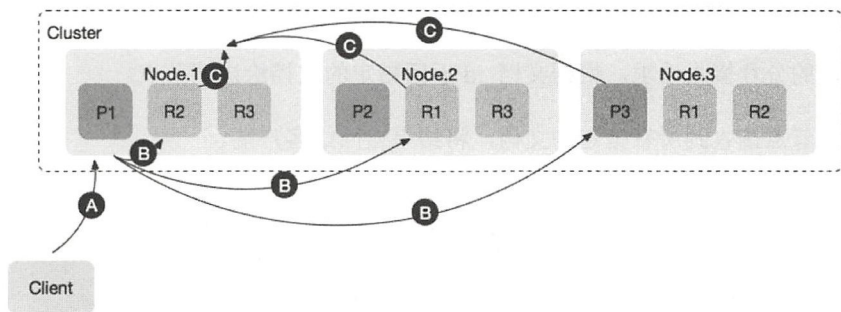


图14-4 查询阶段

Elasticsearch 会通过 `from` 和 `size` 参数对查询结果进行分页。也就是说，每个分片在接收查询请求时，都会查询本地的所有文档，并进行排序，然后再从结果集中找到从 `from` 到 `size` 位置的文档返回给协调节点。而协调节点合并所有分片返回的数据，再进行排序，然后找出从 `from` 到 `size` 位置的文档。所以，每个分片返回给协调节点的结果集都能满足从 `from` 到 `size` 大小的结果集。

默认 `from` 值为 0，`size` 值为 10，也就是返回结果集的前 10 个文档。可以在发送查询请求中自定义 `from` 和 `size` 参数值。比如要查询 `from` 从 10000 开始，`size` 为 1000 的文档，假如有 5 个分片，那么每个分片都会先排序选出自己分片上的前 11000 个文档，并返回给协调节点，那么 5 个节点总共返回 55000 个文档，协调节点再对这 55000 个文档排序，选出从 10001~11000 位置的文档。这是呈指数增长的查询，对系统和服务器都会造成很大的压力，所以建议查询时不要设置比较大的 `from` 和 `size` 值。同时，从 `from` 到 `size` 的值不要超过 `index.max_result_window` 这个索引参数，默认 `index.max_result_window` 的值为 10000。

上面的例子也解释了在设置主分片数时并不是越大越好，越多的分片对于检索文档来说会带来查询性能的下降。同时，我们应该对历史的文档合成分片，减少分片数，在索引时通过路由策略将同类文档放在一个分片中，来优化检索性能。

2. 取回阶段

查询阶段用来标识哪些文档满足客户端请求的要求，而取回阶段才真正将符合要求的文档返回给客户端。

如图 14-5 所示，取回阶段的步骤如下：

A. 协调节点（这里是 Node.1）根据全局结果集中标识的需要返回给客户端的文档在各个

节点中的分布情况，向相关的分片所在节点提交文档请求。

- B. 相关的分片接收请求，根据文档_id 返回指定的文档给协调节点。
- C. 协调节点接收到所有指定的文档，将结果返回给客户端。

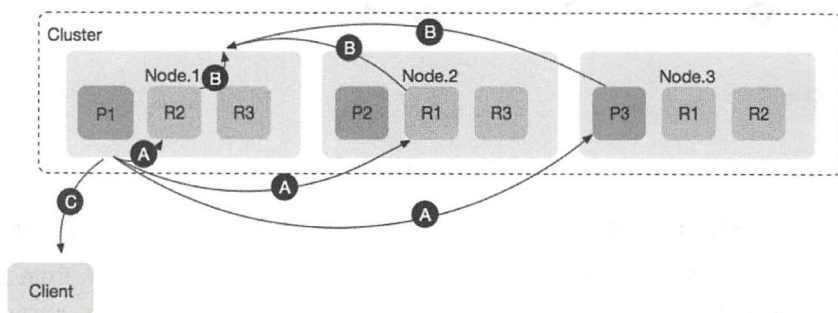


图14-5 取回阶段

14.1.4 分片管理

分片分配是将分片分配给节点的过程，这可能发生在初始化、副本分配、平衡分片、添加或删除节点时。

默认 Elasticsearch 会将所有分片平均分配到所有物理机上，同时会将一个主分片和它的副本分片分布在不同的物理机上，以确保即使一个分片因为物理机故障变得不可用，也依然可以从它的副本分片上获取数据。

1. 分片分配控制

我们可以通过下面的配置参数来控制分片分配的一些属性。

cluster.routing.allocation.enable: 该参数控制允许对哪种分片执行分配操作。

- all: 允许对所有分片执行分配操作。
- primaries: 只允许对主分片执行分配操作。
- new_primaries: 只允许对新索引的主分片执行分配操作。
- none: 拒绝对所有分片执行分配操作。

cluster.routing.allocation.node_initial_primaries_recoveries: 每个节点允许并行恢复的主

分片数量，默认值为 4。

`cluster.routing.allocation.node_concurrent_incoming_recoveries`: 每个节点允许多少传入分片并行恢复，默认值为 2。

`cluster.routing.allocation.node_concurrent_outgoing_recoveries`: 每个节点允许多少传出分片并行恢复，默认值为 2。

2. 分配再平衡控制

当集群需要执行再平衡操作时，可以通过下面的参数来控制。

`cluster.routing.rebalance.enable`: 该参数控制允许对哪种分片执行再平衡操作。

- `all`: 允许对所有分片执行再平衡操作。
- `primaries`: 只允许对主分片执行再平衡操作。
- `replicas`: 只允许对副本分片执行再平衡操作。
- `none`: 拒绝所有分片执行再平衡操作。

`cluster.routing.allocation.allow_rebalance`: 该参数控制允许执行分配再平衡操作的时机。

- `always`: 总是允许。
- `indices_primaries_active`: 所有主分片都处于活动状态。
- `indices_all_active`: 所有分片都处于活动状态。

`cluster.routing.allocation.cluster_concurrent_rebalance`: 该参数控制集群内允许同时执行再平衡操作的分片数量，默认值为 2。如果该参数值设置得较大，则会对系统产生较高的 I/O 负载和较大的网络开销。

3. 分片再平衡触发

当 Elasticsearch 集群中的数据或者节点发生变动时，分片也会发生相应的变化。例如，当在集群中增加一个节点时，集群的拓扑结构将会发生变化，Elasticsearch 将会通过分片分配器 `balanced` 重新调整每个节点上的分片数量。可以通过调用 Elasticsearch API 动态调整 `balanced` 分片分配器的分配策略，各参数配置如下。

`cluster.routing.allocation.balance.shard`: 在节点上分配分片的权重，默认值为 0.45。数值越大，越倾向于在节点层面均衡分配。

`cluster.routing.allocation.balance.index`: 每个索引往单个节点上分配分片的权重, 默认值为 0.55。数值越大, 越倾向于在索引层面均衡分配。

`cluster.routing.allocation.balance.primary`: 在节点上平均分配主分片的权重, 默认值为 0.05。数值越大, 越倾向于在主分片层面均衡分配

`cluster.routing.allocation.balance.threshold`: 如果每个因子与其权重的乘积的总和大于所设置的阈值, 则将触发均衡操作。默认值为 1。

14.1.5 路由策略

在文档索引中, 我们了解到 Elasticsearch 根据文档_id 将文档转发到对应的分片上进行处理, 那么 Elasticsearch 是通过什么策略将文档转发到指定分片上的? 如何保证所有的数据节点都能均匀地保存数据呢? 在执行检索和删除文档的操作时, Elasticsearch 又是如何知道文档所在分片的位置的呢?

其实, 每个文档都会包含 `_routing` 字段, 而 Elasticsearch 正是通过对 `_routing` 字段做 hash, 再对主分片数取模, 得到一个对应的分片号, 再将该文档转发到对应分片上的。用公式表示如下:

$$\text{shard_num} = \text{hash}(\text{_routing}) \% \text{num_primary_shards}$$

而文档的检索也是通过对 `_routing` 字段做 hash, 再对索引分片数取模得到分片号, 然后到对应分片中检索文档的。这也就解释了为什么主分片数在确定后, 就不能再改变的原因。当在已经生成的索引中改变分片的数量后, 之前的路由规则也将发生改变, 这时我们再去检索之前的文档就不能路由到正确的分片中了。

如果在写入数据时没有指定 `_routing` 字段, 系统将默认使用文档的 `_id` 字段作为 `_routing` 字段。可以根据检索需求, 指定一个字段作为 `_routing` 字段, 以此将相同的文档存储在同一个分片中。这对于检索性能的提高有很大的帮助。同时, 在检索文档时, 也可以指定 `_routing` 字段, 那么 Elasticsearch 只会去请求目标分片, 从而让查询更有效率。

请看下面的例子。

以 `user1` 字段作为 `_routing` 字段, 索引文档:

```
curl -XPUT 'localhost:9200/my_index/_doc/1?routing=user1&refresh=true?pretty' -H
'Content-Type: application/json' -d'
'{
```

```
"title": "This is a document"
}'
```

以 user1 字段作为_routing 字段, 检索文档:

```
curl -XGET 'localhost:9200/my_index/_doc/1?routing=user1&pretty'
```

14.1.6 映射

映射是定义文档和它所包含的字段如何存储和索引的过程。例如, 使用映射来定义:

- 哪些字符串字段应该作为全文字段处理。
- 哪些字段包含数字、日期或地理位置。
- 日期字段的格式。
- 为动态添加的字段控制映射的自定义规则。

1. 映射类型

每个索引都有一个映射类型, 来决定如何索引文档。

- meta-fields: 元字段用于定义如何处理文档的元数据。元字段包括文档的_index、_type、_id 和 _source 字段。
- fields 或 properties: 映射类型包含与文档相关的字段或属性的列表。

2. 字段数据类型

每个字段都有一个数据类型, 大致分为以下几种类别。

- 简单类型: text、keyword、date、long、double、boolean、ip。
- 层次结构特性类型: JSON (如对象或嵌套)。
- 专业领域类型: geo_point、geo_shape、completion。

有些字段经常为了不同的目的而需要使用不同的方法索引。比如, 我们可以将字符串字段索引为全文所有的文本字段, 也可以索引为用于排序或者聚合的关键字字段, 还可以以标准分析器、英语分析器或者法语分析器对字符串字段进行索引。这些也是引入 multi-fields 的目的, 大数据的数据类型可以通过设置字段参数来支持 multi-fields。

3. 映射爆炸

在一个索引中定义太多的字段可能会导致映射爆炸, 这可能会引起内存溢出并且很难恢复。

比如，每个新文档的写入都会给索引增加一些新的字段，这些字段会在索引的映射中一直存在，如果数据量小倒不用担心，但是如果有大量的数据，那么随着映射字段的增加，这就会成为一个潜在的危险。所以，我们可以通过下面的参数来限制手动或者动态创建的字段映射的数量，以避免映射爆炸。

- `index.mapping.total_fields.limit`: 一个索引的最多字段数量，默认值为 1000。
- `index.mapping.depth.limit`: 一个字段的最大深度，默认值为 20。
- `index.mapping.nested_fields.limit`: 一个索引的最多嵌套字段数量，默认值为 50。

4. 动态映射

有了动态映射的存在，我们就不需要在使用前定义好字段和映射类型了，新的字段会自动添加到映射中，直接索引文档即可。

5. 自定义映射

虽然动态映射很有用，但是与 Elasticsearch 相比，我们更加了解文档内容，所以也可以自定义索引的映射。我们可以在创建索引时自定义映射，也可以通过 PUT mapping API 将新的字段添加到已存在的索引映射中。

```
curl -XPUT 'localhost:9200/my_index?pretty' -H 'Content-Type: application/json' -d'
{
  "mappings": {
    "doc": {
      "properties": {
        "title": { "type": "text" },
        "name": { "type": "text" },
        "age": { "type": "integer" },
        "created": {
          "type": "date",
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
  }
}'
```

6. fielddata

在默认情况下，大多数字段都能够被索引，这使得它们可以用于搜索。但是，如果需要

字段进行排序、聚合或者从脚本中访问字段值，则需要有不同的搜索模式。

大多数字段都可以使用 `doc_values` 的方式来获取数据，但是文本字段不支持 `doc_values`。文本字段使用 `fielddata` 的方式对字段进行排序、聚合、脚本操作等。`fielddata` 是一种内存数据结构，一个字段只有在第一次被用来执行排序、聚合、脚本操作等时，才会被加载到内存中构建成 `fielddata` 类型的数据结构。它是通过从磁盘上读取每个 `Segment` 的整个倒排索引到 JVM 堆空间中的。在默认情况下，`fielddata` 是被禁用的。

如果开启字段的 `fielddata` 模式，那么对于一个基数比较大的字段来说，它将会消耗大量的 JVM 堆空间。而且，一旦这个字段被加载到内存中，它将在 `Segment` 的生命周期中一直保留。此外，加载 `fielddata` 会导致用户搜索延迟。这些都使得 `fielddata` 默认被禁用。如果需要对一个字段进行排序、聚合、脚本操作等，那么就需要开启这个字段的 `fielddata` 模式。

我们可以通过 PUT mapping API 动态开启一个字段的 `fielddata` 模式。

```
curl -XPUT 'localhost:9200/my_index/_mapping/_doc?pretty' -H 'Content-Type: application/json' -d'
```

```
{
  "properties": {
    "my_field": {
      "type": "text",
      "fielddata": true
    }
  }
}
```

如果需要用到 `fielddata` 的功能，但是又不想占用过多的 JVM 堆空间，则可以通过 `fielddata_frequency_filter` 参数来设置一个过滤频率，只加载小部分的数据到 JVM 堆空间中。过滤频率可以被设置成绝对数值，也可以被设置成百分比。同时，频率是以每个 `Segment` 进行一次计算的，并且只会统计包含指定字段的文档，那些不包含该字段的文档将不会被统计在内。

```
curl -XPUT 'localhost:9200/my_index?pretty' -H 'Content-Type: application/json' -d'
```

```
{
  "mappings": {
    "_doc": {
      "properties": {
        "tag": {
          "type": "text",
          "fielddata": true,
          "fielddata_frequency_filter": {
```



```

        "min": 0.001,
        "max": 0.1,
        "min_segment_size": 500
    }
}
}
}
}
}'

```

7. doc_values

`doc_values` 也用于对字段进行排序、聚合和对脚本操作等，不过它不同于 `fielddata`。首先，它是保存在磁盘上的数据结构，不会占用 JVM 堆空间；其次，`doc_values` 会预先在磁盘上生成，查询时直接使用就行。`doc_values` 以列存储的方式将相同字段存成一列，这对于排序和聚合都是非常高效的。除 `analyzed` 字符串字段外，几乎所有的字段类型都支持 `doc_values`，且 `doc_values` 是默认开启的。如果我们确定在字段排序、数据聚合或者脚本中不使用某个字段，则可以关闭该字段的 `doc_values` 功能，以节省磁盘空间。

```

curl -XPUT 'localhost:9200/my_index?pretty' -H 'Content-Type: application/json' -d'
{
  "mappings": {
    "_doc": {
      "properties": {
        "status_code": {
          "type": "keyword"
        },
        "session_id": {
          "type": "keyword",
          "doc_values": false
        }
      }
    }
  }
}'

```

14.2 可视化工具Kibana

Kibana^[2]是一个开放源码的分析和可视化平台。使用 Kibana，我们可以搜索、查看数据，

与存储在 Elasticsearch 索引中的数据进行交互；执行数据分析，并在各种图表、表和映射中可视化数据。通过 Elasticsearch 的聚合操作，使用 Kibana 可以生成数据的柱形图、线状图、饼图、表格等，共效果分别如图 14-6 和图 14-7 所示。

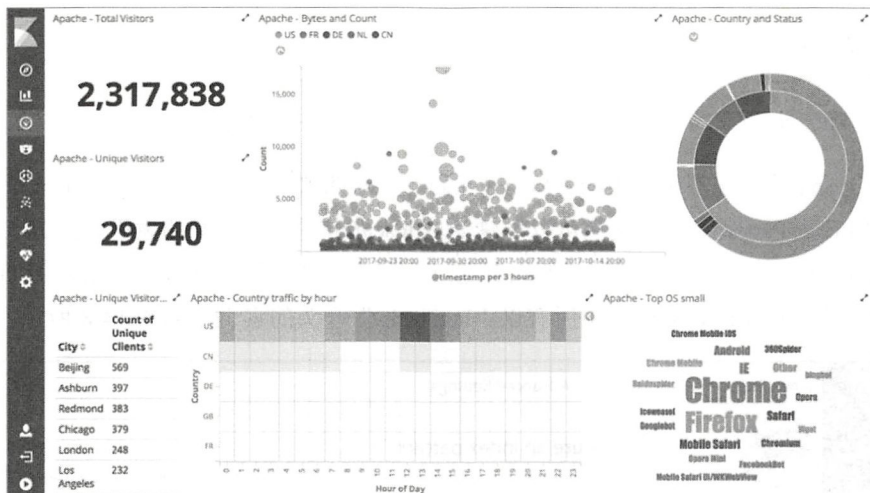


图14-6 Kibana可视化效果（一）

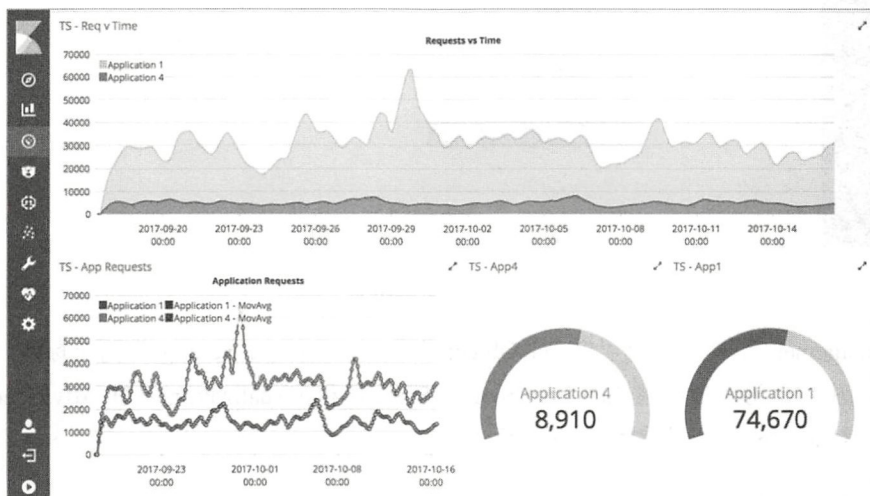


图14-7 Kibana可视化效果（二）

我们也可以通过 Timelion 对 Elasticsearch 中的数据执行时间序列分析。

此外，我们可以通过 Elastic Maps Services 来实现地理空间数据的可视化。利用 Graph 功能

来探索数据之间的关系，以及通过机器学习检测数据异常（Graph 和机器学习功能需要购买收费的 X-pack 扩展来获取）。我们还可以通过 X-pack 扩展来监控 Elasticsearch 集群、Logstash 管道、Kibana 的性能。

14.2.1 Management

在启动 Kibana 后，需要在 Kibana 中关联 Elasticsearch 中已经存在的索引。

点击 Kibana 页面左侧边栏中的“Management”，进入 Kibana 的 Management 页面，选择第一个标签“Index Patterns”，在这里可以创建 Elasticsearch 中已经存在的索引。在 Index Patterns 页面中定义的索引名称可以指定全部的或者通过正则表达式匹配一部分的索引，比如可以通过“logstash-*”来匹配 Elasticsearch 中所有以“logstash-”开头的索引，如图 14-8 所示。

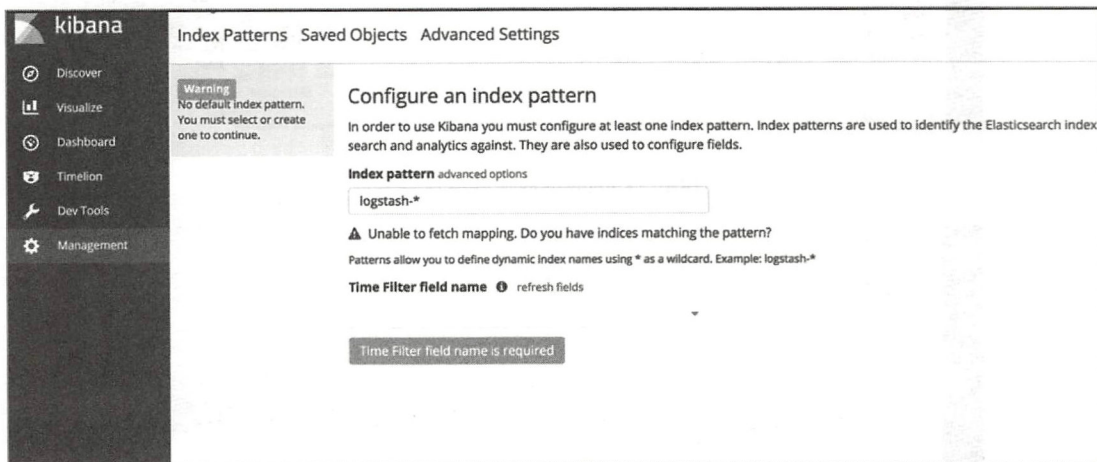


图14-8 新建索引

在 Management 页面上还有“Saved Objects”和“Advanced Settings”两个标签。在 Saved Objects 页面中可以管理自己生成的 Dashboard、Searchs 和 Visualizations；在 Advanced Settings 页面中可以对 Kibana 的一些参数进行设置。

14.2.2 Discover

当 Kibana 关联 Elasticsearch 中的索引后，可以在 Discover 页面中看到索引的日志，如图 14-9 所示。在页面右上方选择时间范围，页面左上角显示为该时间段命中的文档数。其下方为搜索框，可以在这里输入 Elasticsearch 查询语法来搜索想要的数据库。再往下，左下方是文档包含的

字段，右下方就是每篇文档的正文，以及 30 秒粒度的柱形统计图。

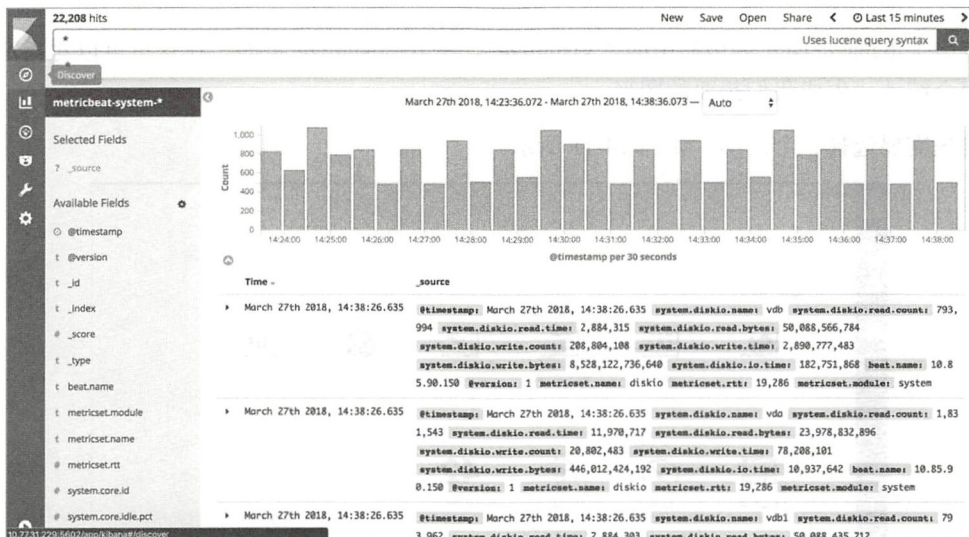


图14-9 Discover页面

比如，想要找出最近一个小时中所有 beat.name 等于 10.75.26.75 的文档。首先调整右上角的时间范围为“Last 1 hour”，然后在搜索框中输入表达式“beat.name:10.75.26.75”，就可以过滤出想要的文档了，如图 14-10 所示。Kibana 默认分页展示前 500 个文档。



图14-10 过滤搜索

14.2.3 Visualize

点击 Kibana 页面左侧边栏中的“Visualize”，进入 Visualize 页面，在这里可以对指定字段进行聚合统计，生成我们想要的图形。

首先需要选择一个图形，如图 14-11 所示。

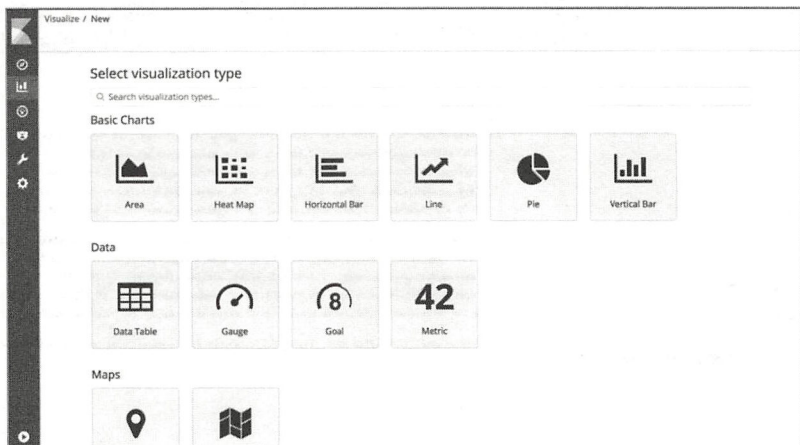


图14-11 选择图形

进入图形数据编辑页面，如图 14-12 所示。Y 轴指定聚合的方式，这里选择 Count；X 轴指定聚合的时间粒度、聚合的维度以及过滤的字段，这里过滤“beat.name:10.75.26.75”，统计每分钟的 Count 数。

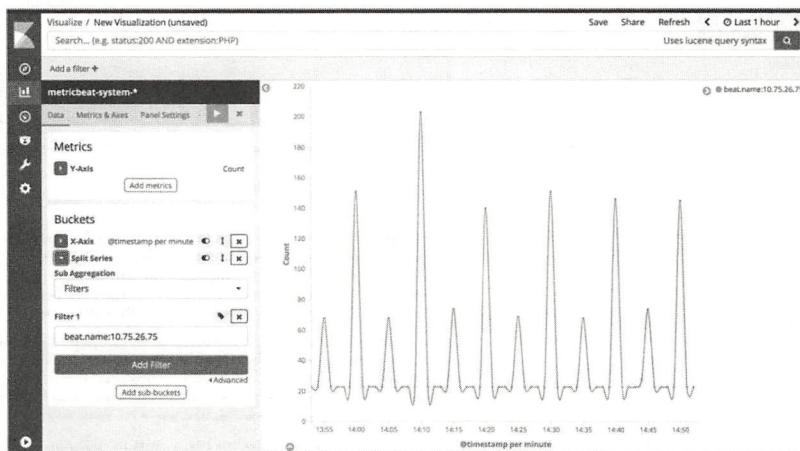


图14-12 图形数据编辑页面

14.2.4 Dashboard

点击 Kibana 页面左侧边栏中的“Dashboard”，进入 Dashboard 页面，在这个页面中可以将 Visualize 页面中生成的图形添加到自定义的 Dashboard 中，如图 14-13 所示。

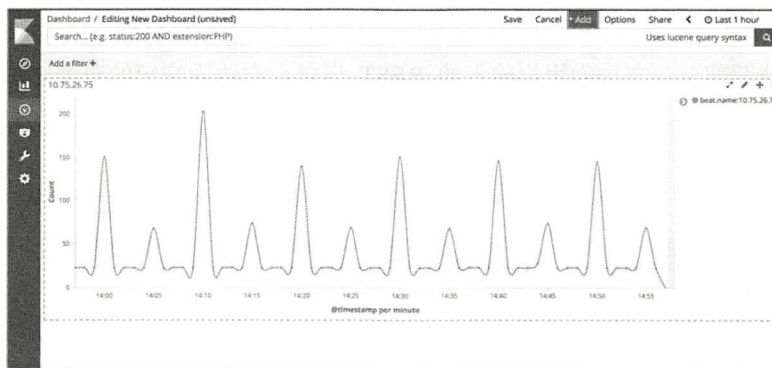


图14-13 Dashboard页面

点击页面上方的“Add”按钮，可以添加更多常用的数据分析图形。点击“Save”按钮，就可以将这个 Dashboard 保存起来以方便每天使用。

14.2.5 Timelion

点击 Kibana 页面左侧边栏中的“Timelion”，进入 Timelion 页面，在这个页面中需要引用 Elasticsearch 自定义的一些函数来聚合过滤指定的文档，生成时序数据图，如图 14-14 所示。



图14-14 Timelion页面

点击页面上方的“Help”按钮，可以查看 Elasticsearch 定义的一些函数的使用方法。在 Timelion 页面中生成的图形，也可以被添加到 Dashboard 页面中。

14.2.6 Dev Tools

点击 Kibana 页面左侧边栏中的“Dev Tools”，进入 Dev Tools 页面，它是一个 Console UI 页面。该页面分两部分，左边为用户输入的 REST 请求，右边为响应的查询结果，如图 14-15 所示。

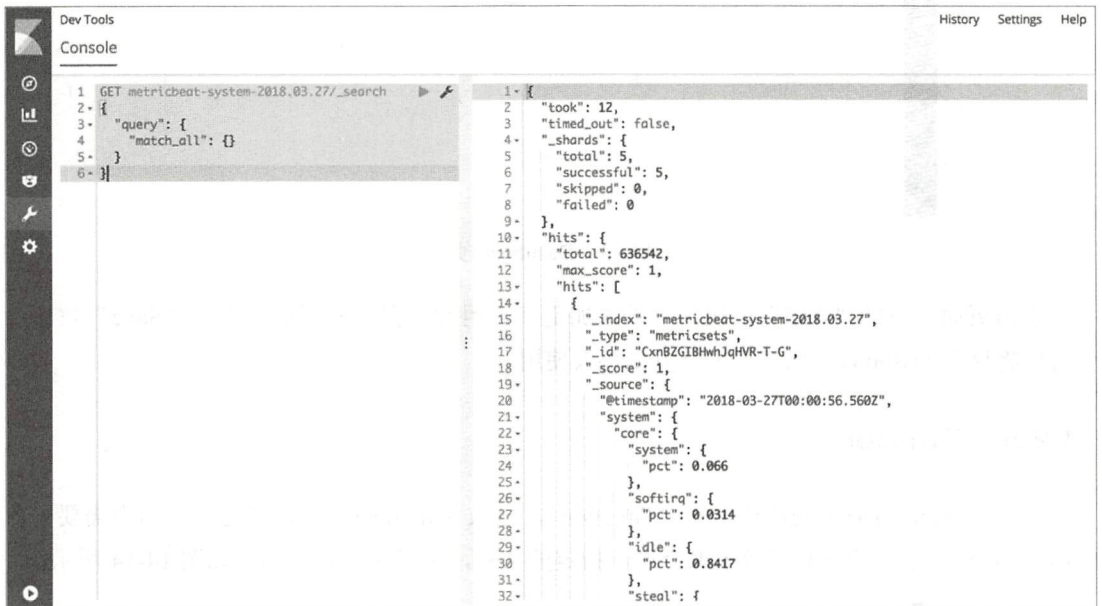


图14-15 Dev Tools页面

Kibana 发展到 6.0 版本，其可视化功能已经非常强大，且可视化页面好看。这里只是简单介绍了主要应用，想要了解更多的关于 Kibana 可视化功能的使用方法，请参考官网：<https://www.elastic.co/guide/en/kibana/current/index.html>。

14.3 ELK搭建实践

14.3.1 Logstash安装配置

1. 环境准备

从 Logstash 官网下载编译好的安装包，无须安装，解压缩就可以使用。地址：
<https://artifacts.elastic.co/downloads/logstash/logstash-6.0.0.tar.gz>。

2. 配置文件

自定义事件流处理配置文件。

```
input {
  kafka {
    bootstrap_servers => "172.16.24.45:9110,172.16.24.46:9110,172.16.24.23:9110,172.16.24.38:9110"
    topics => [ "nginx" ]
    client_id => "192.168.1.1"
    group_id => "nginx_to_es"
    security_protocol => "SASL_PLAINTEXT"
    sasl_mechanism => "PLAIN"
    jaas_path => "/data0/logstash/config/kafka-client-jaas.conf"
  }
}

filter {
  grok {
    match => {
      "message" => %{IP:client} \[%{HTTDATE:timestamp}\] %{WORD:method}
      %{URIPATHPARAM:request} %{NUMBER:bytes:int} %{NUMBER:cost_time:float}
    }
  }
  date {
    match => [ "timestamp", "dd/MMM/yyyy:HH:mm:ss Z" ]
    remove_field => [ "timestamp" ]
  }
}

output {
  elasticsearch {
    hosts => ["192.168.1.4:9200","192.168.1.5:9200"]
  }
}
```



```

        index => "nginx-%{+YYYY.MM.dd}"
    }
}

```

3. 启动服务

通过 `supervisord` 来管理进程，设置 `supervisord` 启动 Logstash 的配置文件。

```

[program:nginx]
command=/data0/logstash/bin/logstash -f /data0/logstash/conf/nginx.conf --http.host
192.168.1.1 --http.port 9600 --pipeline.workers 2 --pipeline.batch.size 1000
--pipeline.batch.delay 60 --path.data /data0/logstash/data/nginx --path.logs
/data0/logstash/logs/nginx --config.reload.automatic
numprocs=1
autostart=true
startretries=3
autorestart=true
user=root
redirect_stderr=true
stdout_logfile=None

```

4. 启动Logstash

执行下面的命令启动 Logstash。

```
supervisorctl start nginx
```

14.3.2 Elasticsearch集群安装配置

1. 环境准备

(1) 从 Elasticsearch 官网下载编译好的安装包，无须安装，解压缩就可以使用。地址：
<https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.0.0.tar.gz>。

(2) 修改 es 用户的文件描述符。

```

[root@local root]# vim /etc/security/limits.conf
es                -                nofile                65536
[root@local root]# source /etc/profile

```

(3) 关闭 swap。

```

[root@local root]# echo "vm.swappiness = 1" >> /etc/sysctl.conf
[root@local root]# sysctl vm.swappiness=1

```

(4) 调整一个进程可以拥有的 VMA（虚拟内存区域）的数量。

```
[root@local root]# echo "vm.max_map_count = 262144" >> /etc/sysctl.conf
[root@local root]# sysctl -w vm.max_map_count=262144
```

(5) 创建 es 用户，Elasticsearch 不允许以 root 用户启动。

```
[root@local root]# /usr/sbin/groupadd es
[root@local root]# /usr/sbin/useradd es -g es -p elastic
[root@local root]# chown -R es:es /data0/Elasticsearch/es/
```

2. 配置文件

(1) 内存调整

如果是源码安装，则修改安装路径下的 config/jvm.options；如果是 rpm 包安装，则修改 /etc/elasticsearch/jvm.options。JVM 堆空间的大小为物理内存的 50%，且不能超过 32GB，32 位系统不能超过 4GB。调整内存有如下两种方式。

○ 通过配置文件设置

```
[root@local root]# vim /data0/Elasticsearch/es/config/jvm.options
-Xms31g
-Xmx31g
```

○ 通过环境变量设置

```
export ES_JAVA_OPTS="$ES_JAVA_OPTS -Djava.io.tmpdir=/path/to/temp/dir"
```

(2) 集群设置

○ 主节点设置

```
[root@local root]# vim /data0/Elasticsearch/es/config/elasticsearch.yml
# Cluster
# 设置集群名称
cluster.name: elasticsearch
# 设置集群信息更新周期
cluster.info.update.interval: 1m

# Node
# 定义节点名称
node.name: "node.1"
# 设置是否为主节点
node.master: true
# 设置是否为数据节点
```

```
node.data: false
# 设置是否为 Ingest 节点
node.ingest: false

# Paths
# 设置数据文件目录位置
path.data: /data0/Elasticsearch/es/data
# 设置日志文件目录位置
path.logs: /data0/Elasticsearch/es/logs

# Memory
# 在 RAM 中锁定 Elasticsearch 的进程地址空间，防止被交换到磁盘上
bootstrap.memory_lock: true

# Network
# 设置绑定的 IP 地址和与其他节点交换的 IP 地址
http.host: 192.168.1.1
# 设置节点间交互的 TCP 端口，在 Elasticsearch 节点之间通过 transport 协议传输数据，默认是 9300 端口
transport.tcp.port: 9301
# 设置是否压缩 TCP 传输的数据，默认为 false，不压缩
transport.tcp.compress: true
# 是否开启 HTTP 协议对外提供服务，默认为 true。由于 Elasticsearch 内部使用 transport 接口通信
# 因此我们可以关闭数据节点的 HTTP 接口，只保留对外提供服务节点的 HTTP 接口，以满足 REST 请求
http.enabled: false

# Discovery
# 定义集群中可用节点的初始列表，当节点启动时使用这个列表进行探测，以便形成集群
discovery.zen.ping.unicast.hosts: ["192.168.1.4","192.168.1.11"]
# 设置形成集群的最少可用的主节点的数量
discovery.zen.minimum_master_nodes: 2
# 开始主节点选举和加入一个集群，节点需要等待的时间
discovery.zen.ping_timeout: 120s
# 故障检测超时时间（通过主节点 ping 其他节点，以及其他节点请求主节点来验证连接是否存活）
discovery.zen.fd.ping_timeout: 120s
# 故障检测超时或失败重试的次数
discovery.zen.fd.ping_retries: 6
# 故障检测的周期
discovery.zen.fd.ping_interval: 30s
```

○ 数据节点设置

```
[root@local root]# vim /data0/Elasticsearch/es/config/elasticsearch.yml
# Cluster
# 设置集群名称
cluster.name: elasticsearch
# 设置集群信息更新周期
cluster.info.update.interval: 1m

# Node
# 定义节点名称
node.name: "node.11"
# 设置是否为主节点
node.master: false
# 设置是否为数据节点
node.data: true
# 设置是否为 Ingest 节点
node.ingest: false
# 设置 bulk 操作的线程池队列大小
thread_pool.bulk.queue_size: 1000

# Paths
# 设置数据文件目录位置
path.data: /data0/Elasticsearch/es/data
# 设置日志文件目录位置
path.logs: /data0/Elasticsearch/es/logs

# Memory
# 在 RAM 中锁定 Elasticsearch 的进程地址空间, 防止被交换到磁盘上
bootstrap.memory_lock: true

# Network
# 设置绑定的 IP 地址和与其他节点交换的 IP 地址
http.host: 192.168.1.11
# 设置节点间交互的 TCP 端口, Elasticsearch 节点之间通过 transport 协议传输数据, 默认是 9300 端口
transport.tcp.port: 9301
# 设置是否压缩 TCP 传输的数据, 默认为 false, 不压缩
transport.tcp.compress: true
# 是否开启 HTTP 协议对外提供服务, 默认为 true。由于 Elasticsearch 内部使用 transport 接口通信
# 因此我们可以关闭主节点、数据节点的 HTTP 接口, 只保留对外提供服务节点的 HTTP 接口, 以满足 REST 请求
http.enabled: false
```



```
# Discovery
# 定义集群中可用节点的初始列表，当节点启动时使用这个列表进行探测，以便形成集群
discovery.zen.ping.unicast.hosts: ["192.168.1.1", "192.168.1.4"]
# 设置形成集群的最少可用的主节点的数量
discovery.zen.minimum_master_nodes: 2
# 开始主节点选举和加入一个集群，节点需要等待的时间
discovery.zen.ping_timeout: 120s
# 故障检测超时时间（通过主节点 ping 其他节点，以及其他节点请求主节点来验证连接是否存活）
discovery.zen.fd.ping_timeout: 120s
# 故障检测超时或失败重试的次数
discovery.zen.fd.ping_retries: 6
# 故障检测的周期
discovery.zen.fd.ping_interval: 30s

# Indices
# 设置每个节点请求的缓存大小
indices.requests.cache.size: 2%
# fielddata 缓存的大小，默认值为堆空间大小的 1%
indices.fielddata.cache.size: 10%
# 设置查询结果的缓存大小，每个节点共享一个缓存，通过 LRU 策略回收，默认值为 10%。该配置属于静态配置，可
# 以在每个数据节点中进行配置
indices.queries.cache.size: 10%
# 控制是否开启查询结果缓存，默认开启。该配置为索引级别，可以为每个索引配置不同的策略
index.queries.cache.enabled: true
# 设置索引恢复时的吞吐量，默认值为 40MB
indices.recovery.max_bytes_per_sec: 40mb
# 设置内存使用达到多少时，触发内存回收，默认值为 JVM 堆空间的 70%
indices.breaker.total.limit: 70%
# 设置 fielddata 使用达到 JVM 堆空间的多少时，触发内存回收，默认值为 60%
indices.breaker.fielddata.limit: 60%
# 设置每个请求的内存使用达到一定值时，触发内存回收，默认值为 60%
indices.breaker.request.limit: 60%
```

3. 启动集群

```
[root@local root]# ulimit -l unlimited
[root@local root]# su es
[root@local root]# /data0/Elasticsearch/es/bin/elasticsearch -d
```

4. 使用API

当集群启动后，可以通过一些 API 来获取集群的状态信息，监控集群异常情况，以及修改集群的参数配置。这里列举一些常用 API 的调用方式，关于详细的使用和其他 API 请参考官方文档。

(1) 查看集群健康状态。

```
curl -XGET 'localhost:9200/_cluster/health?pretty'
```

(2) 查看集群完整的状态信息。

```
curl -XGET 'localhost:9200/_cluster/state?pretty'
```

(3) 查看指定索引和指标的状态信息。

```
curl -XGET 'localhost:9200/_cluster/state/{metrics}/{indices}?pretty'
```

(4) 查看集群索引的状态信息。

```
curl -XGET 'localhost:9200/_cluster/stats?human&pretty'
```

(5) 更新集群范围的参数配置，更改生效时间分为持久性（Persistent）和临时性（Transient）两种。例如，持久性修改索引恢复时的吞吐量。

○ 持久性设置

```
curl -XPUT 'localhost:9200/_cluster/settings?pretty' -H 'Content-Type: application/json' -d'
{
  "persistent" : {
    "indices.recovery.max_bytes_per_sec" : "50mb"
  }
}'
```

○ 临时性设置

```
curl -XPUT 'localhost:9200/_cluster/settings?pretty' -H 'Content-Type: application/json' -d'
{
  "transient" : {
    "indices.recovery.max_bytes_per_sec" : "20mb"
  }
}'
```

(6) 查看集群设置信息。

```
curl -XGET 'localhost:9200/_cluster/settings?pretty'
```

(7) 获取节点统计信息。

```
curl -XGET 'localhost:9200/_nodes/stats?pretty'
curl -XGET 'localhost:9200/_nodes/nodeId1,nodeId2/stats?pretty'
```

(8) 获取节点状态信息。

```
curl -XGET 'localhost:9200/_nodes?pretty'
curl -XGET 'localhost:9200/_nodes/nodeId1,nodeId2?pretty'
```

(9) 删除一个索引。

```
curl -XDELETE 'http://localhost:9200/INDEX_NAME'
```

(10) 关闭索引。

```
curl -XDELETE 'http://localhost:9200/INDEX_NAME/_close'
```

(11) 开启索引。

```
curl -XDELETE 'http://localhost:9200/INDEX_NAME/_open'
```

(12) 判断索引。

```
curl -XHEAD -i 'http://localhost:9200/INDEX_NAME'
```

(13) 查看索引配置。

```
curl -XGET http://localhost:9200/INDEX_NAME/_settings?pretty
```

(14) 获取集群索引信息。

```
curl -XGET 'localhost:9200/_cat/indices?pretty'
```

(15) 查看集群每个节点的分片概况。

```
curl -XGET http://localhost:9200/_cat/allocation?pretty
```

(16) 查看集群每个分片的详细分布信息。

```
curl -XGET http://localhost:9200/_cat/shards?pretty
```

(17) 查看集群指定索引的每个分片的详细分布信息。

```
curl -XGET http://localhost:9200/_cat/shards?pretty
```

(18) 查看指定的索引状态。

```
curl -XGET http://localhost:9200/_cat/indices/INDEX_NAME
```

(19) 查看集群当前数据节点上正在被 `fielddata` 使用的堆空间有多少。

```
curl -XGET http://localhost:9200/_cat/fielddata?pretty
```

(20) 查看集群的各线程状态。

```
curl -XGET http://localhost:9200/_cat/thread_pool?pretty
```

(21) 查看集群指定的线程状态。

```
curl -XGET http://localhost:9200/_cat/thread_pool/bulk?pretty
```

(22) 查看集群等待的任务。

```
curl -XGET http://localhost:9200/_cat/pending_tasks?pretty
```

14.3.3 Kibana安装配置

1. 环境准备

从 Kibana 官网下载编译好的安装包，无须安装，解压缩就可以使用。地址：
https://artifacts.elastic.co/downloads/kibana/kibana-6.2.3-linux-x86_64.tar.gz。

2. 配置文件

修改配置文件 kibana.yml。

```
# 设置 Kibana 启动端口
server.port: 5601
# 设置 Kibana 服务端 IP 地址
server.host: "192.168.1.1"
# Kibana 请求 Elasticsearch 响应的超时时间，默认值为 30000ms
elasticsearch.requestTimeout: 30000
# 设置 Kibana 连接 Elasticsearch 实例的 URL 地址
elasticsearch.url: "http://192.168.1.4:9200"
# Kibana 在 Elasticsearch 中使用索引来存储保存的搜索、可视化和指示板，这里定义索引的名称，默认为 .kibana
kibana.index: .kibana
# 设置 Kibana 没有保存在 Elasticsearch 中的持久数据路径
path.data: /data0/kibana/data/
# 设置 Kibana PID 文件存储位置
pid.file: /data0/kibana/var/run/kibana.pid
# 设置 Kibana 日志文件存储位置
logging.dest: /data0/kibana/logs/kibana.log
```

3. 启动服务

通过 supervisord 来管理进程，设置 supervisord 启动 Kibana 的配置文件。

```
[program:nginx]
command= /data0/kibana/bin/kibana
numprocs=1
autostart=true
```



```
startretries=3
autorestart=true
user=root
redirect_stderr=true
stdout_logfile=None
```

4. 启动Kibana

```
supervisorctl start kibana
```

5. 在Kibana中查询数据

经过简单的 ELK 安装配置，一个功能强大、性能高效的日志监控系统就搭建完成了。只要在浏览器的地址栏中输入 Kibana 网址，这里是 <http://192.168.1.1:5601>，就可以在 Kibana 中搜索从 Logstash 索引到 Elasticsearch 集群中的日志数据了。

14.4 本章小结

Elasticsearch 作为当下最流行的日志检索分析工具，其凭借完善的生态、活跃的社区表现，已经成为运维体系中不可或缺的一环。除具有强大的日志集中检索功能外，将 Elasticsearch 用作时序数据库，在功能上就已经超过了一些传统的时序数据库，如果聚合后的数据被存储为一份单独的索引，那么查询性能也会有很大的提升。

14.5 参考文献

[1] <https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html>

[2] <https://www.elastic.co/guide/en/kibana/current/index.html>

第 15 章

微博广告智能监控系统

计算广告系统是集智能流量分发、投放、结算、CTR 预估、客户关系管理等为一体的大型互联网业务系统，随着微博业务的快速增长，广告系统的复杂度越来越高，成千上万的模块需要不停地进行计算和通信，如何保证这么复杂的系统能够正常、健康运行是一个巨大的挑战。

微博广告智能监控系统依托两大平台：D+（Data Plus）和 Hubble 平台。D+属于商业基础大数据平台，它负责数据采集、存储、计算和分析，并作为底层技术对上层应用提供 API 数据接口。Hubble（哈勃，其含义是数据如浩瀚宇宙之大，Hubble 如太空望远镜，能窥见璀璨的星辰，发现数据的真正价值）平台定位为微博广告智能全景监控、数据透视和商业洞察，与实验平台、效果分析平台、DMP、广告投放后台等共同构成了 D+的上层应用生态。

微博广告 Hubble 平台每日处理 TB 级别的监控数据和万级别的报警规则，Hubble 平台利用机器学习技术进行趋势预测和报警阈值的智能调整，保证商业产品上千台服务器和数百个系统及服务的正常运行。

15.1 背景介绍

15.1.1 监控指标体系

通常，可以用系统处理监控指标的量级（目前已达百万级别）来衡量监控平台的处理能力。这个衡量指标在一定程度上可以反映监控平台的复杂度和能力，但是在实际业务中并不一定需要这么庞大的监控指标体系，很大一部分监控指标是毫无价值或多余的。这些指标要么根本无法真实反映系统或者业务的状态，要么可以直接被其他指标所取代，要么需要结合更多的信息来分析。

总体而言，监控指标有以下几个分类。

1. 机器（系统）指标

机器（系统）指标，主要指从机器资源角度设定的与机器本身比较相关的指标。概括起来，机器（系统）指标包括机器 CPU、Memory、Disk IO / Space、Net IO 等。

监控系统指标的目的在于：发现机器故障、限流与扩容。

2. 应用指标

应用指标分为基础应用指标和非基础应用指标。基础应用指标是指由通用型的应用程序产生的指标，比如 Nginx/Apache 服务器的请求状态以及 access 日志和端口、MySQL 数据库端口、Hadoop 集群运行状态等。非基础应用指标是指非通用型应用程序（业务程序）所产生的指标，比如某个服务的端口号、进程个数等。另外，对于从业务程序产生的日志中抽取出来的指标，也可以归类于非基础应用指标。

3. 业务指标

业务指标是指关注具体业务、产品的指标，如日收入走势、订单数、广告计划数等业务层面的指标。

15.1.2 功能设计原则

在设计系统架构之前，应该先从业务和系统等角度深度挖掘架构要解决的核心问题。对于监控平台而言，可以从平台化、业务和系统架构及设计三个视角考虑来解析核心问题。

从平台化视角考虑，监控报警平台要解决的问题如下。

- 是否能指导 RD 快速定位问题。
- 是否为业务发展的预估提供了参考。

从业务视角考虑，监控报警平台所要解决的核心问题主要有以下几个方面。

- 监控指标：精准性和覆盖率。
- 报警：实效性和准确性。
- 故障诊断。
- 自动处理。

从系统架构及设计视角考虑，监控报警平台要能解决如下几个方面。

- 大数据分析处理能力，包括数据采集、ETL 和数据抽象分析。
- 数据分析处理的实时性。
- 大规模监控指标等时序数据存储、报警规则存储及报警触发。
- 高可用性。
- 数据聚合能力。

15.2 整体架构

基于上述监控指标体系和功能需求，Hubble 平台的整体架构示意图如图 15-1 所示。

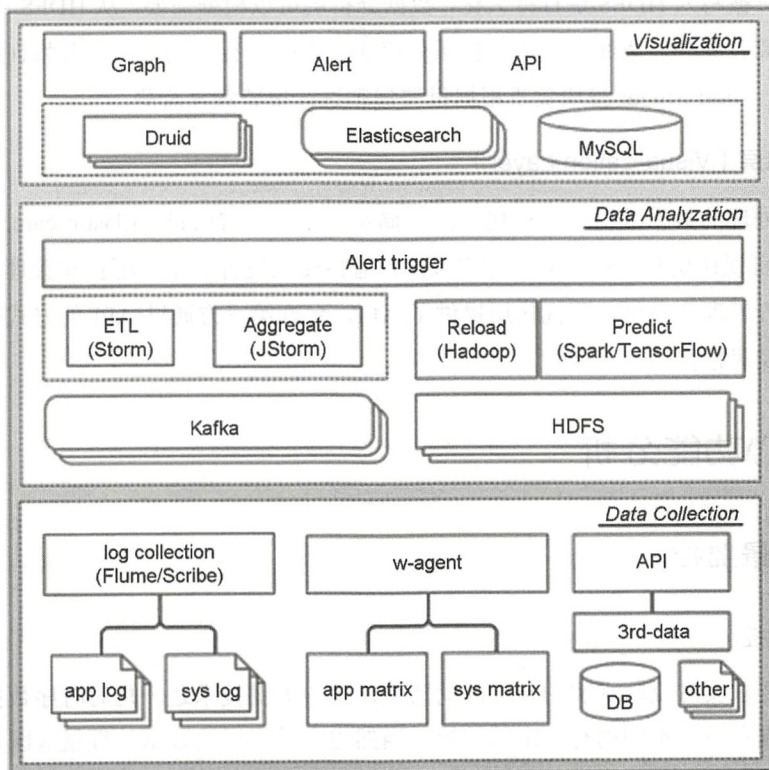


图15-1 Hubble平台的整体架构示意图

Hubble 整体架构包含三个层次。

1. 数据采集层 (Data Collection Layer)

数据采集层负责对系统日志、系统指标、业务日志、业务指标等数据进行实时采集。对于日志数据，支持 Flume、Scribe 等日志收集工具，也支持 Filebeat、Metricbeat 等文件及指标收集工具。对于系统及业务指标，可以通过自主开发的 w-agent 客户端进行数据收集。w-agent 是 Hubble 的轻量级、低资源消耗的数据采集工具，它作为微博广告标准基础工具集的一部分，在服务器初始化时进行安装配置，通过 ZooKeeper 进行配置管理，支持远程更新数据采集配置和配置变更实时生效。同时，数据采集层支持通过 API 的方式直接提交数据，方便数据个性化定制以适应不同的业务需求。

2. 数据分析层 (Data Analyzation Layer)

数据分析层负责将采集到的数据进行 ETL、预处理、分析和聚合。为了提高可用性，采集到的数据将同时被写入 HDFS 进行持久化，数据分析层可以根据需要，从 HDFS 中 Reload 数据并重新进行计算。另外，离线部分的监控预估模块会定时进行模型训练，并将训练后的模型存储在 HDFS 中。Alert trigger 模块负责根据报警规则进行报警触发监测。

3. 可视化层 (Visualization Layer)

经过分析处理的数据被写入可视化层的存储系统中（如 Druid、Elasticsearch、MySQL 及 ClickHouse），可视化层负责根据业务方需求，对监控图表进行展示、配置和管理，以及对报警信息及规则进行管理。另外，可视化层提供了 API，允许第三方通过 API 的方式获取聚合分析后的数据，以及对报警进行管理。

15.3 核心功能分析

15.3.1 全景监控

1. 基础监控

基础监控要求实时反映真实的指标波动情况，其中关键技术之一是对时序数据进行聚合，其聚合粒度根据业务的不同而有一定的差异，当然也会受指标及数据处理量等因素的制约。目前 Twitter、Facebook 等国外公司一般做到 30 秒甚至分钟级别聚合，大部分公司做到 10 秒级别聚合就已经可以满足业务需求了。微博广告监控系统根据业务需求，对部分指标的聚合粒度为 1 秒级别。

基础监控底层使用 D+平台提供实时的数据服务。D+平台是微博广告商业数据基础设施，负责数据收集、存储、监测、聚合及管理，提供高可用的实时流和离线数据服务，在 D+上可以对不同的数据源及实时数据与离线数据进行关联。

D+的整体架构简图如图 15-2 所示。

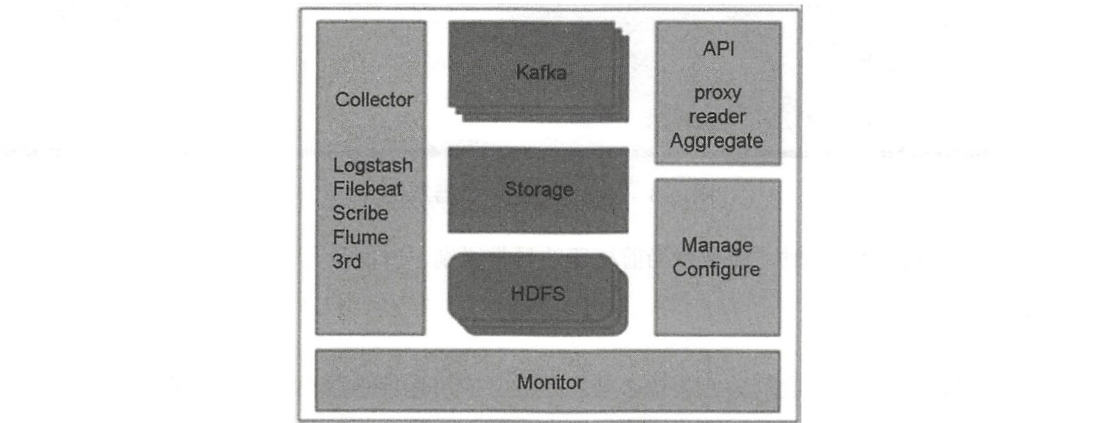


图15-2 D+的整体架构简图

基础指标数据、日志数据会从 D+流出到 ETL 部分进行数据处理，然后输入到 Druid 中提供上层 Graph 展示，同时输出一部分数据（日志）进入 Elasticsearch，以便后续查询。D+位于 Hubble 整体架构中的数据采集层和数据分析层，请参考图 15-1。

2. 服务全景图

假设对监控平台的要求是只允许用一到两个视图来清楚地展现服务监控状态，对所有的需求和指标进行优先级排序，则可以抽取两个关键维度，即机器维度和服务维度。

(1) 机器维度

在机器维度视图下，核心是要清楚地确定所有机器的基本状态，可以简化为健康、亚健康 and 病态三种状态。健康状态表示对机器资源（CPU、内存、网络 I/O、带宽、机器负载等）的使用一切正常，在未来一段时间（如 7 天）内可能也会正常，机器使用者完全不用担心；亚健康状态表示对机器资源的使用存在一定的风险，如高峰期网络 I/O 太高，但仍然未达到影响机器正常运行的程度，机器使用者需要考虑机器 I/O 类型或者扩容，以便应对短期内可能产生的压力；病态是指对机器的某种资源的消耗已经达到上限，如磁盘已满，需要机器使用者立即处理。

根据这样的需求，设计出的视图如图 15-3 所示。

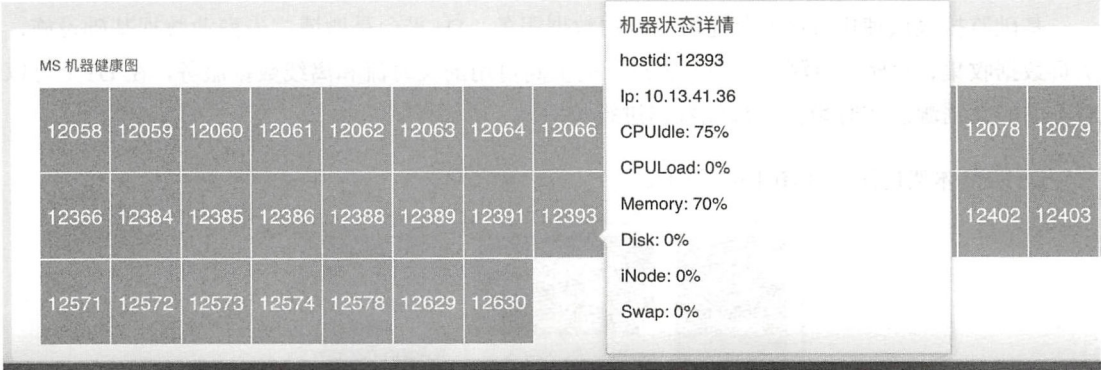


图15-3 服务全景图：机器维度

当某台机器出现异常时，能展示当前机器的异常级别、时间及具体的异常信息。

(2) 服务维度

在服务维度视图下，我们关心的核心问题是业务应用运行是否正常、整个链路是否通畅、是否有异常和告警、出现异常时影响了多少个节点等。并且，可以查看在命名空间下所有关联服务上下游的整体健康状态，针对异常节点展示异常原因和监控报表。

同时，结合自动化平台，对服务之间的上下游关系（拓扑图）进行展示，如图 15-4 所示。

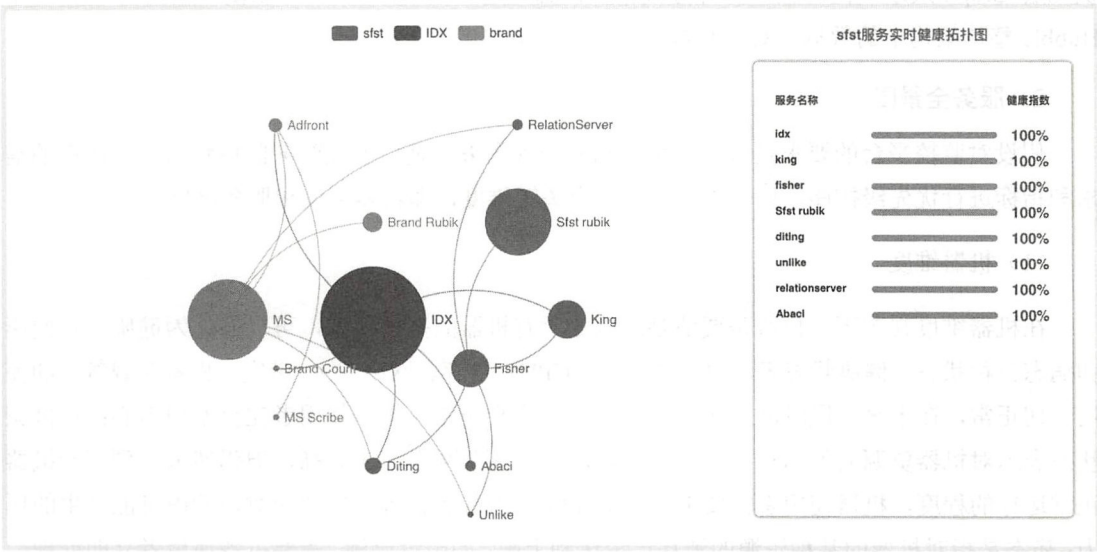


图15-4 服务全景图：服务维度

15.3.2 趋势预测

目前很多企业都在尝试通过趋势预测的方式,提前预测系统或者业务的未来发展状况,未雨绸缪。业内普遍的做法是采用统计学的方法,如 Holter-Winter、ARIMA、3Sigma 等。这类方法的特点是简单,能结合部分历史数据进行趋势预测。然而,它们也有很多不足之处,比如 ARIMA 算法的一个技术难点就是时间序列的平稳化,平稳化的时间序列对于预测结果的好坏起着至关重要的作用。

微博广告团队率先尝试通过机器学习的方法来预测系统指标的变化趋势,取得了一定的效果,目前已经应用于广告曝光量、互动量的趋势预测。

基于机器学习的趋势预测架构示意图如图 15-5 所示。

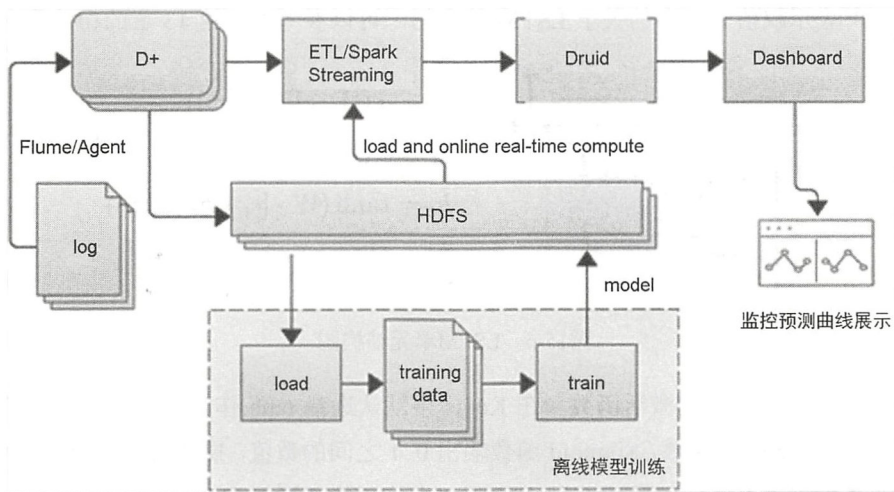


图15-5 基于机器学习的趋势预测架构示意图

该架构主要分两部分:离线模型训练和在线计算。离线部分的数据来源是 HDFS 存储的历史指标数据,输出为模型;在线部分根据模型计算后导入 Druid 中进行存储和部分聚合,通过 Dashboard 进行实时展示。

离线模型训练模块的具体处理流程如下:

1. 监控数据的获取和存储

首先将监控预警平台的监控数据从 HDFS 中提取出来进行数值化。我们在向该模块提供的获取数据接口中发送带有时间戳的 GET 请求,获得了连续 8 天的监控数据作为训练数据集。选

取 8 天这个时间跨度，是充分考虑了监控数据自身可能蕴含的周期性与前后关联性的。

2. 生成模型训练数据集

训练集的窗口长度是指需要几个时间点的值来预测下一个时间点的值。在这里窗口长度为 3，即用 $t-2, t-1, t$ 次的时间间隔进行模型训练，然后用 $t+1$ 次的时间间隔对结果进行验证。数据集格式为： X 为训练数据， Y 为验证数据。我们选取数据集中前 66.7% 的数据作为训练集，后 33.3% 的数据作为测试样本集。

3. LSTM模型结构与参数设置

图 15-6 展示了 LSTM（长短期记忆网络）的一个单元结构。LSTM 单元是预测模型的一个重要组成部分，用于处理时间序列数据。 x_t 表示输入， h_{t-1} 表示上一时刻的状态， h_t 表示当前时刻的状态， W 表示权值。更多的关于 LSTM 的介绍，可以参考本书第 13 章的相关内容。

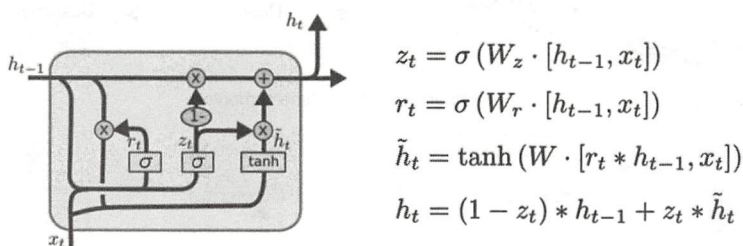


图15-6 LSTM单元结构图

需要确定 LSTM 模块的激活函数（在 Keras 中默认选择 tanh 作为激活函数）；门之间的激活函数一般选择 Sigmoid 函数，Sigmoid 函数输出 0~1 之间的数值，描述每个部分有多少量可以通过，其中 0 表示“不许任何量通过”，1 表示“允许任意量通过”。将接收 LSTM 输出的全连接神经网络（Fully-Connected Neural Network）的激活函数设为 linear（线性）函数；将每一层网络节点的舍弃率（dropout 的值）设为 0.2；使用均方误差（Mean Squared Error）作为误差的计算方式；采用 RMSprop 算法作为权重参数的迭代更新方案，该算法对 RNN 网络通常有较好的收敛效果。

选定模型训练的 epoch（总的训练轮数）为 50 和 batch size（每次训练的样本数）为 1，并在 LSTM 层的输出后面加入一个普通的神经网络全连接层用于输出结果的降维。

4. 模型训练

为了防止过度拟合，将上述获取到的 8 天时间跨度的数据集按 2:1 的比例随机拆分为训练

集和测试集。训练模型后将数据的 X 列作为参数导入模型便可得到预测值，与实际的 Y 值相比较便可知道该模型的优劣。将整个模型训练完成后的参数保存为 h5py 格式（Keras 标准模型格式）的文件，方便在以后的预测或调用中使用。

5. 保存模型结果

将训练完成后的标准的 h5py 格式的 Keras 神经网络模型存入 HDFS 中，用于离线部分数据处理。

6. 结果展示

实时在线部分从 Kafka 中获取实时数据，并使用 HDFS 中训练完成的模型进行计算，将计算结果的数据存入 Druid 中进行图形化展示。

图 15-7 展示了微博广告某产品曝光量（PV）趋势预测曲线效果图。

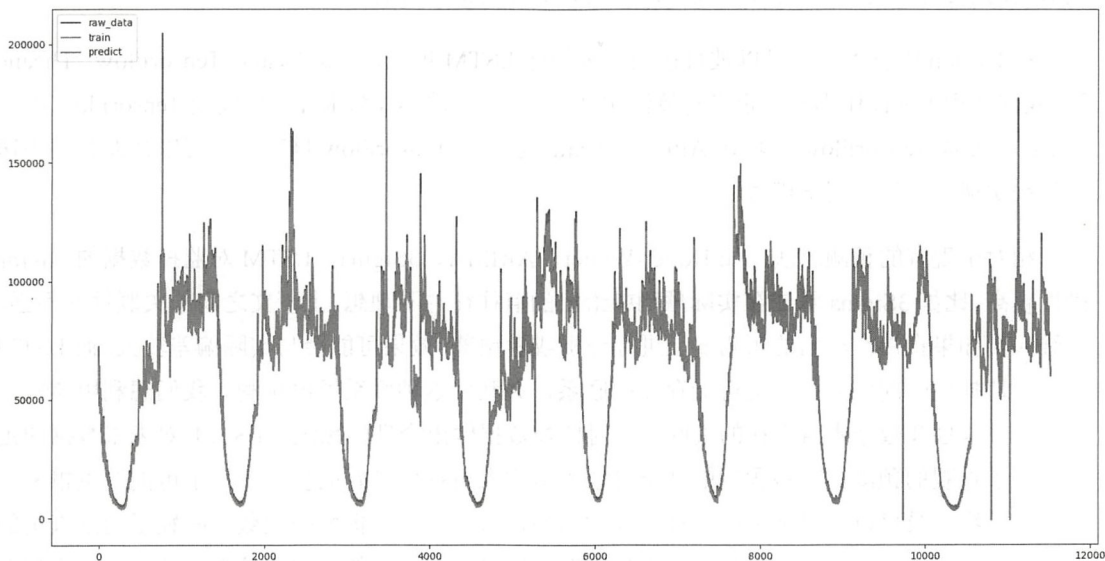


图15-7 趋势预测曲线效果图

（注：图中展示了一周（7天）的数据趋势，其中前5天为训练数据走势情况，后2天为预测数据走势情况）

分析最终得到的效果图，前面（5天）画出了训练数据的趋势曲线，后面（2天）画出了预测数据的趋势曲线，可以从图中看到最终的预测曲线与实际的曲线趋势情况基本吻合，拟合效果非常出色，很好地将原始数据的波峰、波谷以及周期性预测出来了。运维人员可以通过绿色预测曲线来预判监控曲线的整体走势，并且可以了解到可能出现问题的时间点，在可能发生报

警的时间点之前做好准备工作，未雨绸缪，做到心中有数。

时间序列预测是一类相对比较复杂的预测建模问题，和回归分析模型的预测不同，时间序列模型依赖事件发生的先后顺序，同样大小的值改变顺序后输入模型产生的结果是不同的。因此，我们选用 LSTM 模型对监控数据进行预测。LSTM 是 RNN（循环神经网络）的变体，LSTM 的特点就是在 RNN 结构基础上添加了各层的阀门节点。这些阀门可以打开或关闭，用于判断模型网络的记忆态（之前网络的状态）在该层输出的结果是否达到阈值，从而加入当前该层的计算中。阀门节点利用 Sigmoid 函数将网络的记忆态作为输入进行计算，如果输出结果达到阈值，则将该阀门输出与当前层的计算结果相乘作为下一层的输入；如果没有达到阈值，则将该输出结果遗忘掉。每一层包括阀门节点的权重都会在每一次模型反向传播训练过程中更新。LSTM 模型的记忆功能就是由这些阀门节点实现的。当阀门打开时，前面模型的训练结果就会关联到当前的模型计算，而当阀门关闭时之前的计算结果就不再影响当前的计算。因此，通过调节阀门的开关，我们就可以实现早期序列对最终结果的影响。

在 Python 中有不少包可以被直接调用来构建 LSTM 模型，比如 Kears、TensorFlow、Theano 等。我们选用 Keras 作为模型定义与算法实现的机器学习框架，将 Keras 集成到 TensorFlow 中，让 Keras 变成 TensorFlow 的默认 API。让 Keras 运行在 TensorFlow 框架上，可以帮助我们快速搭建和实现一个神经网络模型。

相对于现有的预测算法，如 Holter-Winter、ARIMA、3Sigma，LSTM 对监控数据的预测准确性较高。比如 3Sigma 算法在实际预测中给出的值往往并不理想，上下文之间的关联性并不强，运维人员如果直接在给出的预测值上进行分析或者预警，效果可能会与实际偏差较大。而 LSTM 能很好地抓住时间序列上下文可能存在的联系，并利用这种联系做出预测。我们想利用该模型的前后关联性和数据本身存在的关联性，对监控数据做出合理的预测。LSTM 对未来数据的走势给出了直观的预测，趋势预测的结果可以为运维人员提供良好的参考，对于可能发生报警的情况，运维人员可以较早地采取有针对性的措施，有效地减少报警的次数，减轻了运维人员的工作负担。基本达到了不漏报和不误报的平衡，同时根据大量的历史监控数据训练出的模型有很强的泛化和预测能力，不用实时地加入数据更新训练，也不需要耗费人力、物力进行长期维护。后期的工作只需要将更新的模型替换到预测模块中即可。我们可以将节省下来的监控的人力成本，投入到智能监控的研究当中。

图 15-8 展示了微博广告某产品曝光量预测值与实际值偏差占比分布情况。

可以看到，对于 PV 预测值与实际值小于 1000 的占比约为 73%，小于 1500 的占比约为 96%。按照 1000 次曝光（PV 量）转化为广告计量单位为 1CPM，1.5CPM 内的误差占比为 96%。

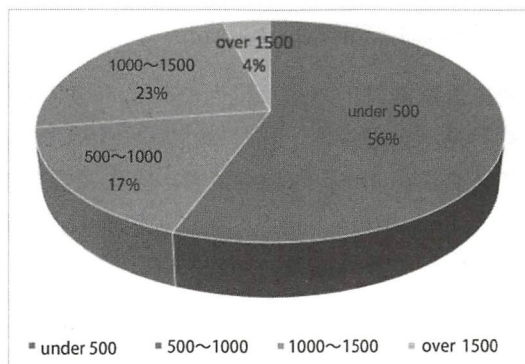


图15-8 微博广告某产品曝光量预测值与实际值偏差占比分布情况（单位：PV）

需要注意的是，目前基于机器学习的趋势预测还不能很好地支持对精度要求非常高的指标，如按点击收费的广告互动数指标。

15.3.3 动态阈值

系统报警往往结合监控来做，对于某个指标触发所设定的阈值后向相关人员发送消息提醒。目前大多数做法是固定阈值，这个阈值是根据经验设定的，其优点是简单、直接、操控性强。其缺点是经验值不准，如果 GAP 设置得过小，则会增加报警次数，导致误报率增加；如果 GAP 设置得过大，又会导致漏报。另外，很多数据指标的波动呈周期性变化，通过经验或者人工很难设定一个合适的 GAP。为此，可以结合趋势预测技术，在预测的基础上增加动态阈值。比如报警条件为趋势预测曲线上下 10%，这个百分比可以根据业务进行调整。

15.3.4 服务治理

要想让一个系统变得健壮，具有高可用性，除需要完善的监控分析、及时的报警策略等被动方式外，还需要主动发现系统中可能存在的隐患。特别是在微博这样的平台中，无法预知什么时候会有流量高峰，更无法预知这个高峰会有多高。比如一个热门事件就可能导致流量瞬间飙升，如 2017 年鹿晗、关晓彤公布恋情，微博广告流量在 1 分钟内就飙升了近一倍，即使是动态扩容的速度也赶不上这样瞬间突发的流量高峰。所以，面对这样的场景，如何把控系统，对系统的每个服务、每个节点的承载能力及瓶颈、缺陷都了然于胸，才能及时封堵缺口，快速响应故障。

1. 环境选择

流量压测的方式有很多种，可以对单台机器、单个服务进行流量压测，也可以通过将流量集中转发到线上一小部分服务来提高单台服务器的 QPS，还可以搭建一个和线上环境一模一样

的系统。但是上述方法都不能模拟生产环境的真实情况。

单台机器和单个服务的流量压测对于测试单机性能和单个服务能够承载的最大 QPS 是可以的，但是这样的结果永远都只是一个理论值，往往评估出来的结果过高。因为它忽略了上下游服务的依赖，如果一个服务被要求在 10ms 内响应请求，它自身需要消耗 2ms 的执行时间，但是下游响应它的时间却超过了 8ms，这样，最终结果在上游看来都是超时的。也许所有服务压测出来的结果都能满足要求，但是服务与服务之间的网络质量又有可能成为瓶颈。

将流量集中转发到线上一小部分服务来提高单台服务器的 QPS，这样的方法对整个系统一部分服务节点的流量评估可能是合理的，但是不能忽略了服务后端访问的存储类资源，很多时候我们并不能做到对存储类资源也集中流量。

至于搭建一个和线上环境一模一样的系统，依然没法确保服务之间的网络质量都是一样的，也没法确定会不会有一些其他因素可能影响流量评估的结果。更何况对于很多公司来说，在同样的机房中，具有相同的机器数量，拥有相同的软硬件配置和环境，搭建一个测试环境并不是一件容易的事情。

所以，综合来看，只有以线上环境为测试环境，将测试流量直接导入线上环境中，才能真正模拟流量高峰下系统的可用性，无论是前端服务还是计算节点，抑或是后端存储类资源，都可以真实展现它们在流量高峰下的性能和问题。虽然这样做是一件很冒险的事，但是如果我们做好足够的准备工作，小心翼翼地进行，那么就可以将风险降到最低。

2. 流量压测/容量评估

GoReplay 是一款用于捕获和重放实时 HTTP 流量的开源工具，可以不间断地用线上实时流量来测试系统性能。

采用 Go 开发的 GoReplay 部署简单，只需下载一个可执行文件就可以在生产环境中的服务器上捕获指定端口的流量并转发到测试环境中，或者添加一个测试流量的标识，再重新发送到生产环境中，如图 15-9 所示。

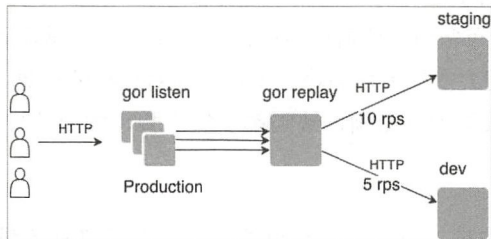


图15-9 GoReplay流程示意图

通过 GoReplay 转发 HTTP 请求示例如下：

```
./goreplay
# 从 80 端口捕获 HTTP 请求
--input-raw :80
# 设置自动退出时间
--exit-after 60s
# 将请求转发到 192.168.1.1 的 80 端口
--output-http "192.168.1.1:80"
# 将 10%请求转发到 192.168.1.2 的 8080 端口
--output-http "192.168.1.2:8080|10%"
# 自定义测试流量标识
--http-set-header 'User-Agent: Gor'
# 指定超时时间
--output-http-timeout 100ms
# 输出请求状态
--stats
--output-http-stats
```

3. 性能评估

可以通过 GoReplay 将线上流量复制一份或者多份，然后导入线上环境中，线上环境通过 HTTP Header 来区分真实流量和测试流量。接下来要做的就是在线上系统流量翻倍的情况下，通过全链路 Trace 系统来观察各个服务节点的性能瓶颈。

全链路 Trace 系统是通过在每个请求中增加全局唯一的 Trace ID，在服务与服务之间增加 Span ID、Timestamp 等信息，来全局跟踪一条流量处理链路的。通过全链路 Trace 系统还可以统计 QPS 在各个服务节点的变化，以及每个节点处理请求的耗时占比。

通过全链路 Trace 系统，可以很直观地发现在流量高峰下，各个服务节点处理请求的 QPS、耗时分布、状态码分布等指标信息，为下一步的系统优化提供可靠的数据依据。

15.4 本章小结

监控系统首先要解决的问题是指标覆盖率，大而全，然后在此基础上进行深耕，提高准确性，最后是精简和抽象，发现数据后面最本质的东西。微博广告基础架构团队在监控方面有一定的积累，并率先在监控中利用机器学习技术进行了一些尝试，未来将在智能预测和服务自动化处理（降级、恢复等）方面进行更深入的尝试。

第 16 章

微博平台通用监控系统

几年前，在开始建设微博平台监控系统时，参与者们面临几个选择，一是在原有的系统上改造。比如在新浪内部就有一套自主研发的、运行很久的监控系统，但是它只有系统级别（CPU、内存、网卡之类）的监控，没有业务监控。一般这样的内部系统，由于人员流动，再加上文档缺乏，已经很难维护和开发了。

二是选择业界主流的、成熟的监控系统，比如 Zabbix、Zenoss 等。它们的优点很明显，比如都有成熟的、适应各种操作系统的 Agent，都集成了相对完善的规则和告警策略，但其存在的问题是二次开发成本太高，扩展性不好，数据量越大越不好控制。

在这里要感谢笔者的前同事引入了 Graphite 这个体系，它很快在业务监控这个领域推广开来。原因很简单，就是得益于其灵活性。在一个需求变化非常快的环境下，比如互联网行业，需求是不明确的（或者说很难预计），很多监控系统的数据和展现绑定在一起（在不太复杂的、变化较少的业务环境下这是优势，入门门槛低），当需求变化时，会导致数据模型、存储格式、传输格式等发生一系列变化，从提出需求到完成功能通常耗时很长，这在追求速度的互联网行业是不可接受的，某些成熟的、商业化监控系统甚至根本无法进行这种改动。

新浪内部基于 Graphite 的业务监控系统，从几万指标逐步扩大到几千万指标，Dashboard 数量从几十个增长到数万个，其提供的功能也从简单的 Dashboard 扩展到了应用 API、手机 App、监控大厅九宫格等，支撑各个业务部门的日常巡检、故障定位、告警、性能分析、压测、动态扩缩容以及容量分析等。

16.1 背景

对业务监控的需求，最初是由微博平台研发 Feed 部门提出来的，它隶属于“微博平台全面防御体系”规划，大意就是包含架构、监控、容量、干预四个部分，对微博平台的业务进行全面掌控，如图 16-1 所示。

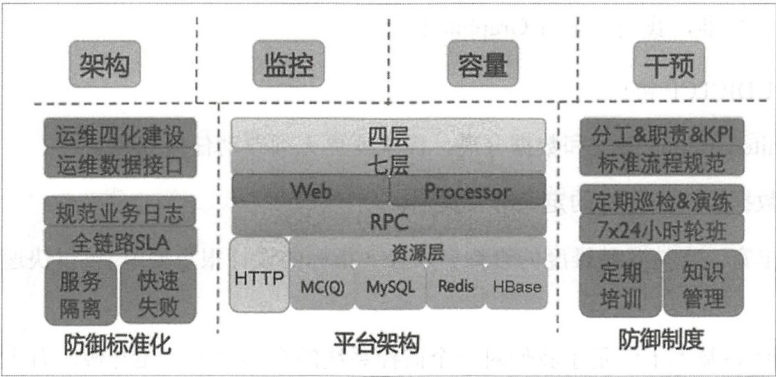


图16-1 微博平台全面防御体系示意图

其中监控和架构是最先开始建设的，我们引入的 Graphite 体系，为监控大厅提供了微博整体业务的业务量、错误数等较宽泛的指标监控。

业务发展迅速、需求变化多端、功能日新月异带来的问题如下：

- 需要一个实时反映业务运行状况的系统，但定制开发的监控系统总是滞后，需求得不到及时满足。
- 监控系统一般都关注系统，很少基于业务来做，对接口的监控也是出于报警的目的，只关注重点，收集信息不全面。
- 基于全量日志的大数据分析，虽然可以得出部分关键指标，但一般都是非实时的、离线的，属于事后印证而不是实时决策，且过于宏观和笼统，照顾不到细节。
- 历史上各类“监控系统”都将展现和数据混在一起，一旦需求和应用场景发生变化，通常会重构部分或整个系统，投入产出比很低。

针对这些问题，监控系统的解决方案应该具有以下特征。

- 需要全方位、无死角地收集业务运行状态，覆盖要全面。

- 必须是实时的，不能是离线计算和批处理，实时系统才可能支持动态调配资源和决策。
- 必须是轻量级的，不需要传输大量日志到后端，无须进行复杂的配置，要提供丰富的函数，支持聚合、汇总和正则匹配。
- 对系统的侵入性足够小，无须安装复杂的客户端，或者在代码层面做重大改动。
- 无须开发，只要按照协议发送数据，即可自动在后端生成可视化图表。

按照上面的标准，我们再来看 Graphite 的特性。

- 支持 UDP/TCP 协议。
- Graphite/Grafana 展现和数据分离，需求变更无须改动任何代码。
- 明文数据格式，指标的定义非常灵活。
- 函数丰富，支持各种维度的查询、聚合（指标的设计很重要），可以快速满足不同的需求。

Graphite 体系基本上满足了一个监控系统的全部要求，还开源，有大量的周边应用和小工具，方便集成和二次开发。

就像我们之前所提到的，在“大工具、小系统，小工具、大系统”的选择中，我们一开始就选择了用“小工具”来构建“大系统”。主要原因如下：

- 需求不明确/变化快，优秀的系统和架构是逐渐演化出来的，不是一开始就设计好的。
- 业务方技术栈多样化，接入方式必须简单。
- 快速迭代，所见即所得。
- 成本要低（在数据传输量、带宽占用、系统资源占用等方面）。

16.2 整体架构

微博平台监控系统整体架构示意图如图 16-2 所示。

这个看上去挺大的系统，处处都显出“小”——没有任何一个组件需要同时完成两件事，同时，任何一个组件都可以随时被替换为其他解决方案。

总体上，系统分为数据采集、数据路由、聚合运算、数据分发、数据存储、告警、API、数据可视化和第三方应用。

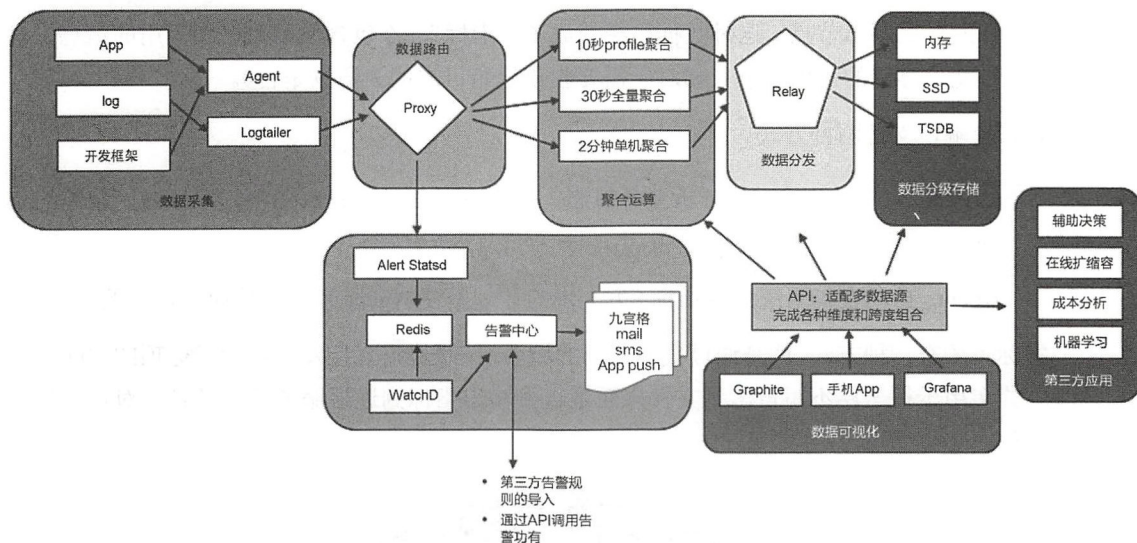


图16-2 微博平台监控系统整体架构示意图

16.3 核心模块

16.3.1 数据采集（Logtailer）

我们采集的数据（指标）主要包括以下几种。

- 系统指标：CPU、内存、网卡流量、负载等。
- 服务指标：Nginx、Tomcat、RPC 框架等的 QPS、响应时间。
- 性能数据：接口性能（QPS、响应时间）、各阶段耗时、依赖资源性能、slowtop。
- 异常数据：各种错误代码、栈信息、报错信息等。
- 业务数据：AppKey 访问量、话题阅读量等。

这样的监控平台属于大数据平台吗？

- 从源数据量的角度来看，数千万指标需要秒级存储和运算，每天至少需要处理几百亿条日志数据，也算“大”数据了。
- 从结果数据量的角度来看，经过采集端规则化和聚合（1 秒周期）后，按照协议组成指

标发送到路由节点，这时数据已经变成了“指标”，不包含原始数据和无用的信息，对带宽、存储的占用很小（相对于需要收集原始日志的大数据平台）。

在采集端，支持两种方式。

- Logtailer: 以 tail 的方式分析业务日志，提取有用信息，完成初步的聚合后，按照约定的协议发送到路由节点。
- Agent: 自动收集系统监控指标（下一步的计划是使 Agent 也能支持业务日志的分析）。

实际上还有一种方式，就是通过我们公开的域名，按照协议直接发送包含指标项的 UDP 包。这种方式多用于框架层不写日志到文件，而是直接输出指标到远端服务器的场景。对业务进行抽象，如图 16-3 所示。

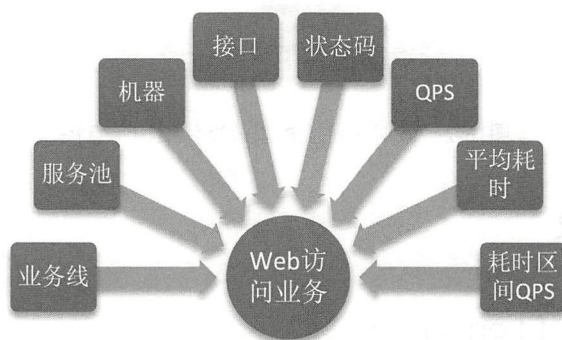


图16-3 业务抽象

也就是说，以上这些信息基本上可以描述一个业务的状态。所对应的 key（指标）设计如下。

(1) http_2xx, http_4xx, http_5xx, 响应时间区间分布数据

- dpool_sso.ilogin_tc.http_2xx(4xx,5xx).get_user_id.hits
- dpool_sso.ilogin_tc.http_2xx.get_user_id.interval1(interval2,interval3,interval4,interval5)

(2) 平均响应时间

- dpool_sso.ilogin_tc.http_2xx.get_user_id.mean

(3) 单机数据

- dpool_sso.ilogin_tc.byhost.172_16_xx_xx.http_2xx(4xx,5xx).get_user_id.hits

(4) 总量数据

○ dpool_sso.ilogin_tc.http_2xx(4xx,5xx).get_user_id.hits

指标里面已经包含了业务线、服务池，接口、IP 等信息，这为后面的聚合、分发提供了必要条件（这只是基本的 key 设计，还可以添加更多的信息，但 key 的长度会受到一些系统的限制，比如 MC 不容许 key 超过 255 个字节）。

目前系统支持以下几种针对 key 的运算和处理协议，如图 16-4 所示。

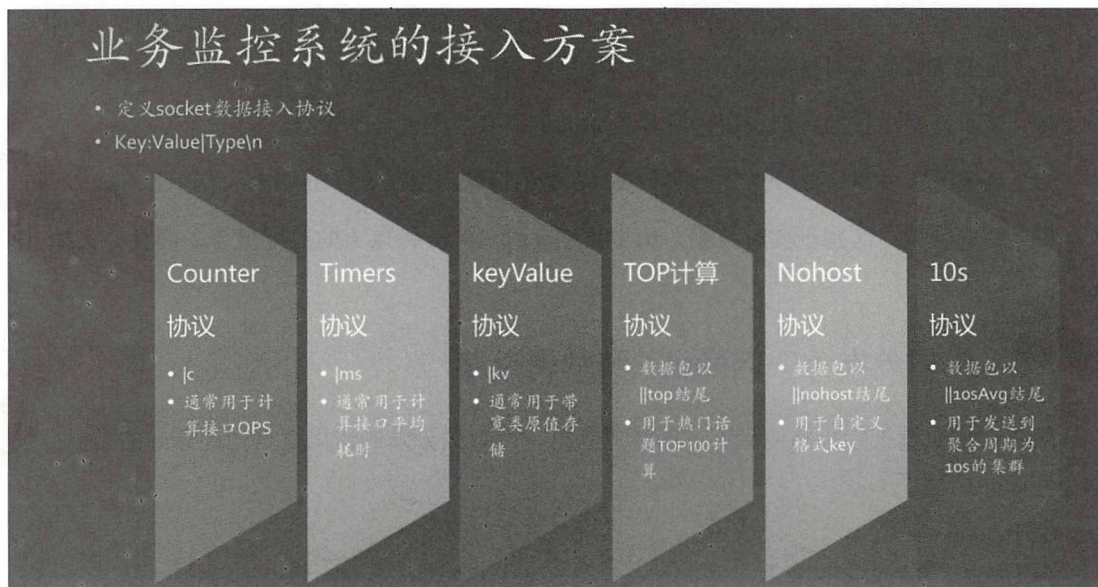


图 16-4 系统支持的运算和处理协议

理论上，任何部门只要按照协议发送数据，就算接入监控系统（在实际应用中，会有白名单、反垃圾、审计等安全性方面的设计，防止无效数据和滥用），门槛非常低。

在这里需要注意的是，我们的系统不提供“日志查询”，那是大数据平台的工作。

16.3.2 数据路由（Statsd-proxy）

数据路由组件的作用如下。

- key hash：在一个聚合周期内，一个 key 必须固定落在一个节点上。
- 不同的计算类型和聚合周期，对应着不同的计算节点。

- 不同用途的指标，需要路由不同的集群（比如告警和监控分流）。

除了上述作用，还有一些特殊用途，比如核心数据的备份、多机房冗余，都需要数据路由组件参与复制、分流。

16.3.3 聚合运算（Statsd）

对指标的计算和操作分为以下几种。

- 实时数据：求和、求平均，同时又区分 10 秒、30 秒、60 秒不同的运算周期。为兼容压测接口的实时性要求，还提供了 1 秒一次的运算模型。
- 聚合计算：按照接口、IP、服务池和机房等多个维度自动聚合。
- 函数运算：按照运算周期进行各种流式函数的运算，比如 P99、Baseline、慢速比、TOPN 等。
- 历史数据：区别于实时数据，历史数据会做 10 分钟以上的粒度聚合，而且会去掉 IP 信息。

16.3.4 数据分发（C-Relay）和数据存储

数据路由负责把指标路由到不同的计算集群，而数据分发的作用就是把计算好的指标分发到不同的存储节点上。

```
cluster realtime
  carbon_ch replication 1
    10.39.xx.xx:2003
    10.39.xx.xx:2003
    10.39.xx.xx:2003
  .....
```

数据分发示意图如图 16-5 所示。

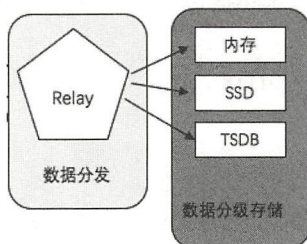


图16-5 数据分发示意图

为了提高访问性能和写入效率，我们对指标数据进行了存储分类。

(1) 实时数据，存储介质：内存

考虑到这类数据的使用频率最高（聚合周期为 1~10 秒，时间跨度为 2 小时至 1 天），访问效率高于一切，所以我们采用了内存文件系统，指标就保存在时序数据文件 Whispser 中，并将整个存储目录 mount 到内存中。内存的读写性能比磁盘高很多，这样基本解决了磁盘 I/O 问题；受限于内存的容量，实时数据的保存周期一般较短

(2) 准实时数据，存储介质：SSD 磁盘

这里说的准实时，其实不是指数据有延迟，而是指数据使用频率较低、聚合周期较长（比如 1 分钟，时间跨度为 2~7 天），一般都用在几天内同比、环比等场合。这类数据对效率的要求没有那么极端，但是数据时间跨度较长，对性能仍有一定的要求。所以，对于这类数据我们选择保存在 SSD 磁盘上（仍然是 Whispser 文件），兼顾容量和访问效率。

(3) 历史数据（7 天以上），存储介质：TSDB

这类数据就是典型的“冷数据”，只有在特定的分析场合才会被用到，访问频率最低。但其数据量却很大，存储周期可能有数年，那么既支持海量数据存储，又具有时序数据特点的存储方式就是使用 TSDB（OpenTSDB 基于 HBase）。不同于 Whispser，TSDB 需要单独的存储/访问接口。

为了数据的安全性，实时和准实时数据会在 TSDB 中持久化备份一份，但会定期清除过期数据。

而完成对存储分发的就是 C-Relay 这个工具，它会按照存储时长、前缀、通配符等各种方式完成对指标的分发，同时具有按照 key 进行垃圾数据过滤的功能，并支持正则表达式。

```
match ^stats\.timers\.openapi\.*\.[0-9]{6,}\..*
  send to blackhole
stop
```

这条配置信息表示一旦满足条件，这种类型的 key 就会被发送到 blackhole，和/dev/null 是一个意思。

16.3.5 告警模块

告警模块展示如图 16-6 所示。

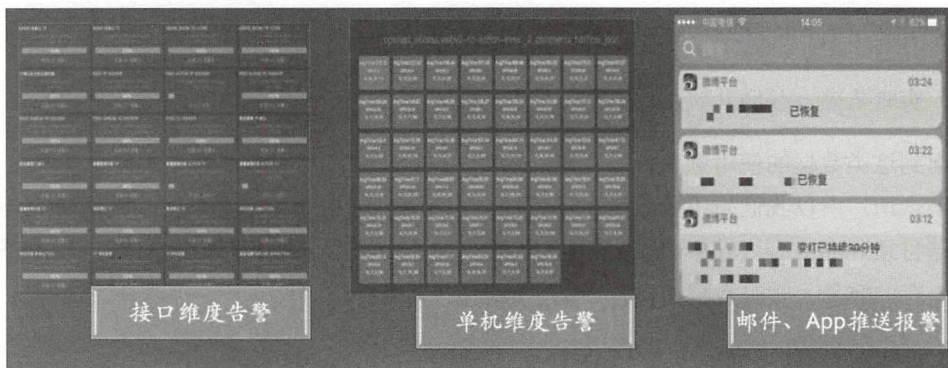


图16-6 告警模块展示

我们提供了多种告警途径，有适合 Service Desk、监控大厅使用的九宫格（颜色和声音），也有方便移动办公的手机 App 推送。关于告警设计也经历了多个发展阶段。

1. 轮询式告警

业务方配置好需要告警的接口、阈值和告警方式，我们通过 Crontab 运行一组脚本，定期扫描存储（Whisper 和 TSDB）中的相关指标，满足条件即触发告警，如图 16-7 所示。

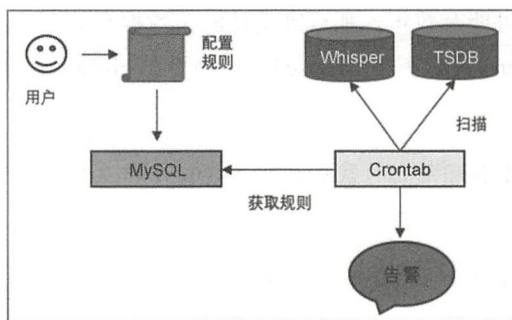


图16-7 轮询式告警流程图

这种方式的原理和配置比较简单，在指标相对较少的情况下可以使用，指标一旦多起来，光配置告警规则的工作量就十分巨大，此外还存在以下几个问题。

- 延迟：从 Whisper 和 TSDB 中轮询指标，这意味着已经有了一段时间的延迟（时间长短取决于聚合周期、存储步长，以及计算流程上的时间损耗）。
- 灵活性差：系统是动态发展的，业务量在不断增长，固定的规则、阈值很难适应这种情况，网络抖动引起的短暂异常也会引发大量的误报。

2. 流式告警

流式告警解决了轮询式告警的一些缺点。一方面，虽然还少不了有一些人工配置告警的工作，但已经可以做到根据推送数据自动提取告警规则，在很大程度上简化了告警配置；另一方面，告警数据不再存储，而是经过流式计算后存入 Redis 队列，满足阈值和触发条件的会被 WatchD 消费（告警），不满足的会被定期从队列中清除，如图 16-8 所示。

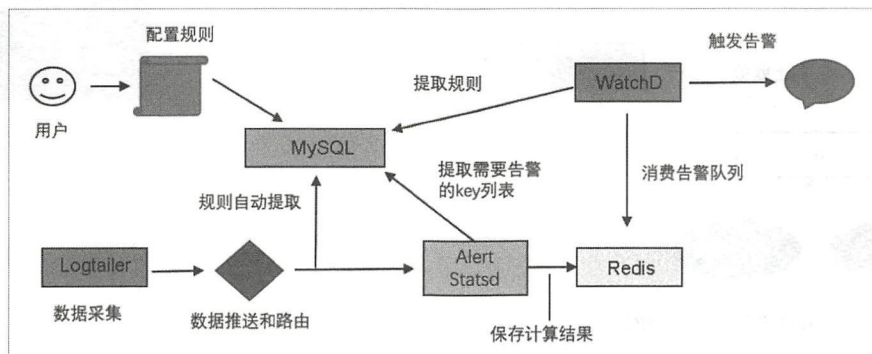


图16-8 流式告警流程图

流式告警有如下几个好处。

(1) 实时性：数据流过就会计算是否符合规则，不用再走存储流程。

(2) 规则抽象：我们按照几个层次（业务线、服务池、IP、接口）抽象了告警逻辑。

○ 最小级别的接口异常标准按照 SLA 或者 KPI 指标自动设定。

○ 逐层向上收敛。在服务池级别有多少 IP 发生异常即触发本级别的告警事件（比如 5% 的 IP 响应时间超过 SLA 设定，为服务池 warning 事件；超过 10% 为服务池 error 事件），到了业务线级别，告警的判断标准是有多少服务池发生异常（warning, error, fatal 等）。每一层的告警事件都会向上收敛，如果不超出上一层的 SLA 设定，则只会展示在九宫格中或者事件列表里，而不会直接报给用户，最终向上汇总的最高级，用户在宏观层面既可整体把握可用性（不影响整体的话可以不用立刻干预），又可逐级向下追溯具体的问题节点，如图 16-9 所示。

(3) 规则自动提取：只要推送数据符合协议（包含业务线、服务池等信息），报警规则和收敛方式就会自动进入配置库（MySQL），后面的九宫格、邮件、推送都自动生成，不需要人工干预（这里生成的都是抽象出来的通用规则，用户也可以通过后台定制自己的专属规则）。

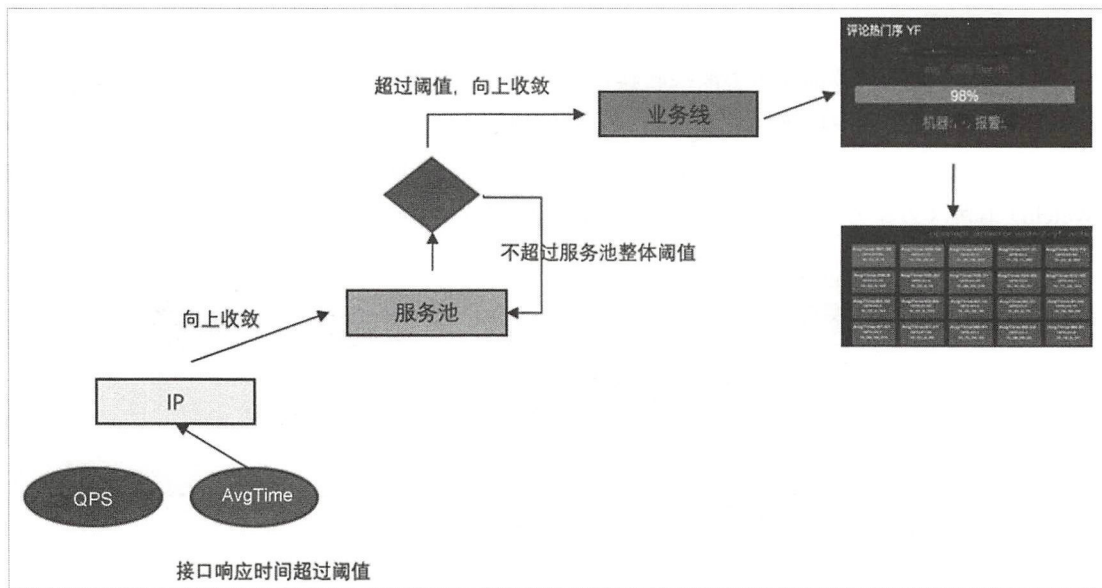


图16-9 基于SLA的告警收敛示意图

(4) 告警 API：除针对单个指标设置规则外，还可以通过 API 批量导入告警规则，尤其是在指标特别多的时候（在某些情况下，业务方对告警逻辑的抽象和我们不同，因此不适用自动提取规则）。同时，因为我们有手机 App push 推送服务，有需求的业务方可以通过 API 调用 push 推送服务。

流式告警上线后，基本上自动规则就能解决 80% 的告警需求，但仍有一些特殊化的需求还不能覆盖，比如陡增突降告警、同比环比等复杂的需要经过统计运算的告警。另外，九宫格虽然可以让 Service Desk 24 小时监控整体服务，但毕竟它是基于 SLA、经过高度抽象的，开发和运营人员关心到这个层次就可以了，但运维人员不同，他们需要细节，需要追溯到具体的问题点。

这个阶段我们可以称之为“重服务质量，轻告警”，真正需要通知到用户的告警数量比较少，大部分告警事件都因为对整体服务影响不大而被“降噪”了。

3. 智能告警

智能告警阶段是通过算法，对告警的数量和质量（误报率）进行不断修正的过程。在智能告警阶段主要的方法包括：

- 将告警聚合成“关联事件”。

- 合并重复的告警。
- 自动恢复策略。
- 告警分类。

16.3.6 API设计

API 设计主要有下面几个用途和考虑。

- 多数据源、各种格式的匹配。
- 多维度和时间跨度的查询组合。
- 黑名单和非法请求的过滤。

API 设计示意图如图 16-10 所示。

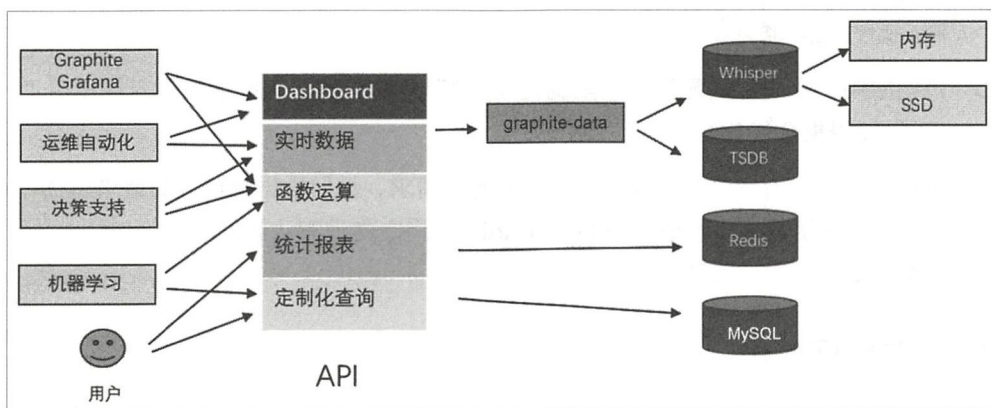


图16-10 API设计示意图

为了提高存储效率，我们之前设计了“时序数据”的多级存储，基于 Whisper 的数据无论是在内存还是 SSD 中都可以通过 Carbon 访问，但 TSDB 是另外一个体系，有自己的接口，且 Graphite 并不支持 TSDB（Grafana 支持 TSDB 数据源，但 Graphite 用户群也很大）。所以，我们又增加了一个称作“graphite-data”的中间件，用来适配 Carbon/Whisper 和 TSDB 两个数据源，以后或许可以扩展到其他存储方式。

除了上面说的“时序数据”，还有几类数据，如报表数据（适合关系数据库存储）、TOP 类数据（实时计算代价太大，可预处理）、缓存/队列数据（一般用来生成 Dashboard），这些数据不太适合“时序数据”库处理，它们通常会被存储在 MySQL、Redis 中，用户访问这类存储，

就需要 API 来做不同的适配。

还有一类“实时数据”，不做任何存储，直接聚合计算完成后输出结果，一般用在压测、流量监测、自动扩缩容方面。这类数据的聚合周期可小到 1 秒，API 支持直接在 Statsd、Relay 上获取，省略了后面的存储流程。

增加 API 后，对不同类型的请求（实时数据、历史数据、报表数据等）、不同时间跨度的请求进行分流和隔离，避免耗时较长的“慢”查询拖慢整个系统，单个存储或者数据源的失效，不会影响其他正常请求；对于某些来源的大量请求，可以做到限流。

API 黑名单和非法请求的规则如下。

- 超时规则：任何一条请求都必须在规定的时间内完成，超过时间则直接断掉，并进行记录。当超时次数达到阈值后，此条请求会被直接拒绝，5 分钟后恢复。
- 条目限制：有些极端的请求可能会返回数十万个指标，对此我们有最大返回条目的限制，超过上限也会被当作非法请求予以记录。
- 规则限制：在指标通配符中，限制使用“*”号的个数，有些层级不容许使用“*”号（可以使用集合代替）。
- 审计功能：每天会出一份报表，列出最慢的请求、返回数据量最大的请求 TOP100 等信息，还有访问最多和最少的 Dashboard、一周之内没有访问的数据等指标，方便我们对数据进行分类优化和处理。

16.3.7 数据可视化

在总结前些年的实践经验后，我们一直坚持数据和展现分离，所以在可视化方面做到了以下几点。

- 将 Dashboard 交给 Graphite/Grafana，不做任何定制图表的开发。
- 告警规则和九宫格自动产生：对业务的抽象程度足够高，所以绝大多数业务都能用同一套告警逻辑来适配，接入新业务和新数据不用改动代码。
- 提供功能强大的 API：有特殊需求的第三方可以通过它定制自己的数据展现，进行二次处理和分析数据等。
- 多端同步：我们还提供了 Graphite/Grafana 手机端的适配，方便用手机 App 访问 Dashboard。

很多监控系统的失败，一个重要原因是数据和展现不分离，所以在数据展现、可视化方面，我们还是重点依赖 Graphite/Grafana 的自有功能，以及自动化流程（九宫格告警），超出此范围，则建议业务方通过 API 进行二次开发和定制。

16.4 第三方应用

既然我们提供了很好用的 API，那么业务方可以发挥的地方就有很多了（毕竟他们是程序员）。下面介绍几个基于 API 的第三方应用。

16.4.1 决策支持系统

如图 16-11 所示为决策支持系统界面。

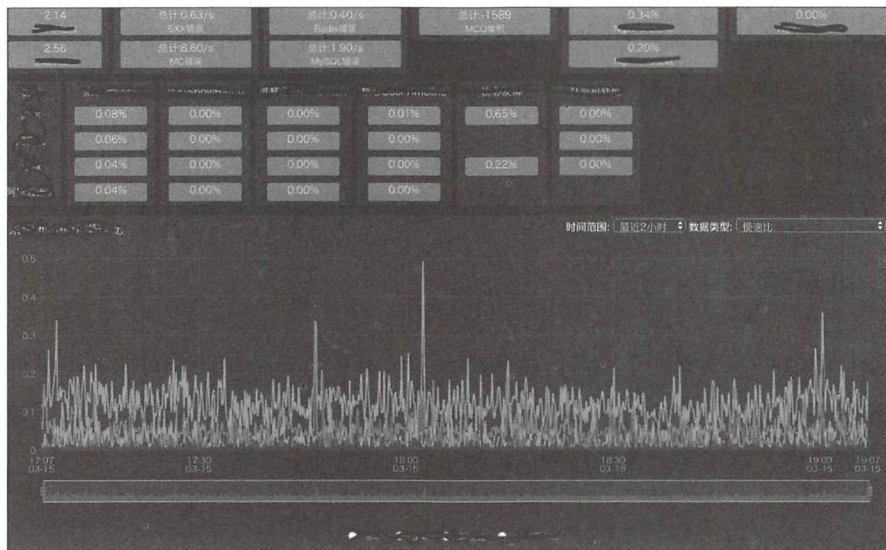


图16-11 决策支持系统界面

这是典型的降低维度、归一化的应用，上百万个指标、数千个 Dashboard、上万台服务器，最终可以抽象成一个页面（有些地方是单位内部信息，已做了处理）。每个业务都可以用单一指标来衡量，最终都归一化为 0~1 之间的一个数字来表示健康状况。

所有的数据都是通过 API 获取后二次加工而成的。

16.4.2 运维自动化

通过 API，微博平台的运维人员也根据监控数据开发出了不少的自动化工具。

- 自动化扩缩容：根据计算出来的容量水位线（上限和下限），从混合云平台动态申请和退还云服务器。
- 自动 503 错误处理（503 指 Service Unavailable，是一种 HTTP 状态码，当服务器出错时通常返回 503 错误）：根据负载、性能状况，摘除某些出现 503 错误的节点。
- 自动流量切换：根据流量分析结果，优化流量路由。

16.4.3 成本分析和容量日报

在所采集的数据中有一个 AppKey 维度，用来标识微博平台的调用方身份，根据这个维度来聚合访问量，就能很容易计算出每个调用方对平台的资源消耗比例。

容量日报则按日统计线上服务池、业务线的当前容量，为是否需要扩容和采购提供依据，日报依赖服务器流量、负载、访问量、压测结果等多项指标进行计算。

16.4.4 机器学习

我们设计的监控系统有一个业务方的保障系统，其中有一个功能是对业务流量进行探测，检测流量是否突变（就是异常检测），通过 API 定期获取很多天的历史数据，用于数据训练，生成流量期望模型，再用实时数据与期望模型进行对比学习。

16.5 本章小结

微博平台所构建的是一套通用型监控系统，该系统需要对接更多的业务方，需要处理各种异构的日志数据，监控指标数量也相对较多，因此对系统的通用型架构设计要求较高。本章详细介绍了微博平台的监控系统核心架构设计原理，以及构成通用型监控系统的核心模块和基本功能。最后介绍了包括决策支持系统、运维自动化等的第三方应用，这些应用是智能化运维的核心组件，一个设计精良的监控系统能够非常容易地对这些组件进行集成。

附录A

中国大数据技术大会 2017（BDTC 2017）

CSDN 专访实录

我们采访了大会推荐系统论坛的讲师微博广告技术专家彭冬，他给我们带来了题为《微博商业化大数据平台从 0 到 1 架构演进及应用实践》的分享，以下为正文。



彭冬：微博广告技术专家，目前主要带领基础架构团队，负责商业基础大数据平台（D+）、数据可视化平台（Hubble）、智能运维体系等广告基础设施建设，关注大数据、人工智能、智能运维等方向。

CSDN：请向大家介绍一下你自己和目前所从事的工作，以及正在关注哪些技术领域？

彭冬：2011 我年加入微博广告团队，先后负责过品牌广告和竞价粉丝通广告等多个微博商业化产品的架构设计和核心功能开发。目前主要带领基础架构团队，负责商业基础大数据平台（D+）、数据可视化平台（Hubble）、智能运维体系等广告基础设施建设，我们正在关注大数据、人工智能、智能运维等方向。

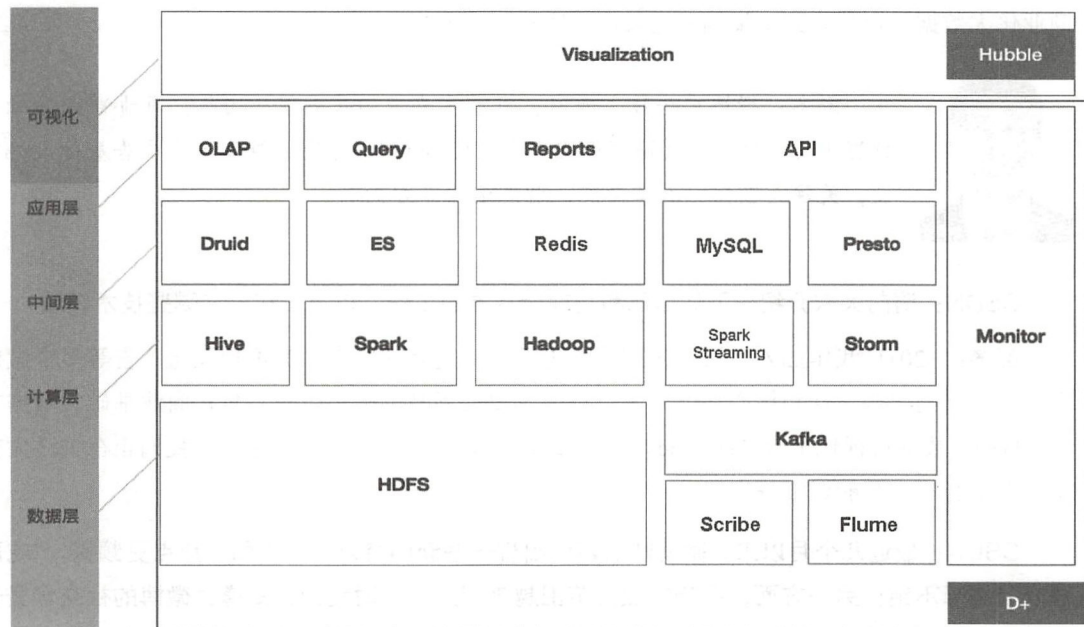
CSDN：最近几个月以来，微博热点事件出现一些新的特点，一方面，热点更频繁，“吃瓜群众”络绎不绝；另一方面，影响的业务范围越来越广。同时我们也知道，微博的社交场景也极其复杂，随着数据量的日益增长，微博的大数据平台正面临着什么样的技术挑战？

彭冬：热点事件经常出现在微博上，也体现了微博非常强的社交属性及其传播特性。当热门事件发生时，用户活跃度剧增，沉睡的用户也会被唤醒，流量会在短时间内出现成倍的增长，会给整个系统带来极大的挑战。对于大数据平台而言，数据存储部分主要是数据规模的突增，

以及网络 I/O 的突增对集群的考验，计算部分尤其是实时计算部分需要更多的资源，此时我们通常会通过一定的智能调度策略优先保证核心计算资源。在数据分析层面，因为主要使用离线计算，对实时性要求不高，可以适当避开高峰对资源的争抢。

CSDN：目前微博商业基础大数据平台（D+）的架构是怎样的？

彭冬：在这里也简单讲一下微博商业基础大数据平台（D+）的定位和设计目标。微博从 2011 年开始进行商业化尝试，逐渐形成了非常完善的广告产品生态，如果说商业化的第一个阶段主要以新产品形态的开拓为目标的话，那么微博商业化的第二个阶段应该是商业产品的精细化。精细化尤其是依托大数据平台来构建商业数据分析的上层决策会显得至关重要。D+ 定位于建立商业化数据（与商业化有关的一切数据）的完整生态，其包含数据采集、存储、数据仓库和数据建模、计算（离线和实时）、数据可视化和数据 API 平台的整个体系。其设计目标是作为最基础、最底层的基础平台，支撑任何业务方对商业数据的分析需求。D+ 的整体架构技术栈如图 A-1 所示。



图A-1 D+的整体架构技术栈

简单来说，D+分为数据层、计算层、中间层和应用层。D+通过 API 可以与上层更多的应用进行组合，比如监控系统和实验平台等。

CSDN: 可否简单分享一下D+的架构演进过程?

彭冬: D+经历了大致三个阶段的发展。

(1) 0.1 版本 (2011—2015 年)

这个版本主要是将各种异构数据进行整合与搜集,通过 Scribe 搜集所有的广告业务数据、日志数据到数据中心,为监控平台提供了数据分析能力,同时建立了简单的数据报表功能。

(2) 1.0 版本 (2016—2017 年)

我们发现, Scribe 作为数据搜集工具经常会有严重的日志堆积问题,在 1.0 版本中我们引入了 ELK 套件,对 Filebeat 进行了二次开发,同时发布了微博广告版本的日志采集工具 Farmer。另外,我们将 Scribe 替换成 Kafka,引入了多种数据聚合与关联技术。对于时序数据,我们引入了 Druid 作为引擎,为监控平台、实验平台、查询引擎提供实时的数据存取架构。同时,我们在日志搜索这块使用了 Elasticsearch,对全量数据构建索引。广告主报表引擎则基于 OLS (微博内部实时流计算引擎) 来构建。

(3) 2.0 版本 (2017 年至今)

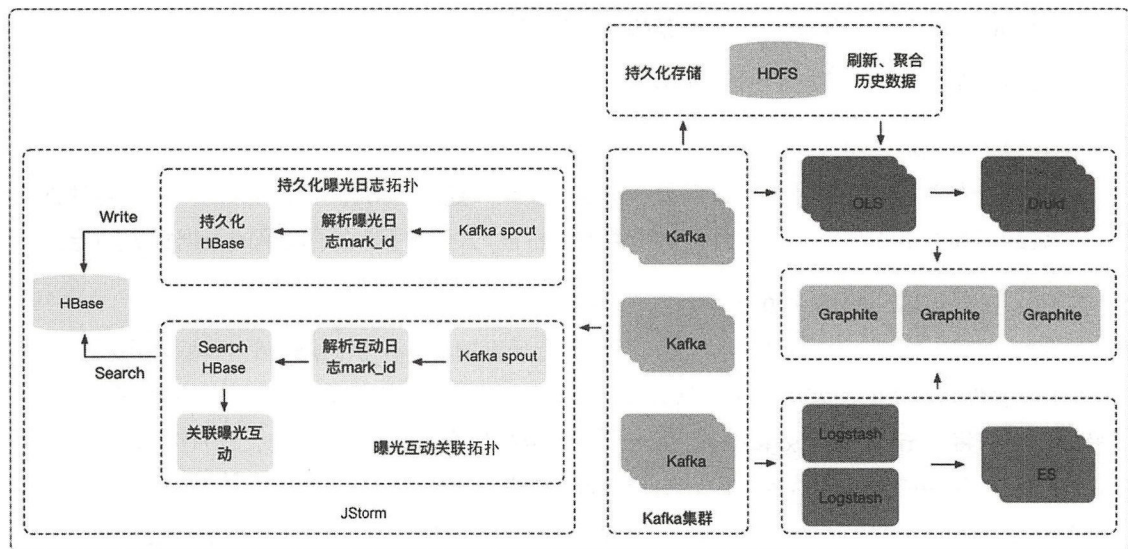
在 2.0 版本中,我们对 D+做了重新定位,D+作为底层基础数据平台,为各个业务方提供数据存取、数据分析、数据挖掘等基础数据支撑。我们构建了分层的数据仓库系统,建立了基于业务需求的数据集市。同时,在实时计算部分引入了 Flink。为了提供高效的数据分析能力,我们构建了 OLAP 平台,在 D+上支持 Ad-hoc 查询。

CSDN: 微博广告系统Hubble也是极其强大的,它是一个秒级大规模分布式智能监控平台。目前Hubble的核心功能是什么? 解决了哪些核心问题?

彭冬: 我们知道 Hubble 就如太空望远镜,能窥探浩瀚宇宙(大数据)的价值。我们希望 Hubble 能够作为一个开放的数据可视化平台,为商业数据分析、数据计算等提供可视化功能。Hubble 1.0 最初是构建在从数据采集到实时处理再到可视化的一个智能监控平台上的,随着 Hubble 的不断发展, Hubble 2.0 已经超越了当时仅仅支撑智能监控平台的定位,目前专注于数据可视化,而底层数据部分则托管于 D+平台。我们的智能监控平台也在不断发展演进,不仅体现在对大规模时序数据的实时监控上,在动态阈值、故障预测、异常点检测等智能运维(AIOps)方向也有一定的实践和尝试。

CSDN: Hubble的整体架构是怎样的?

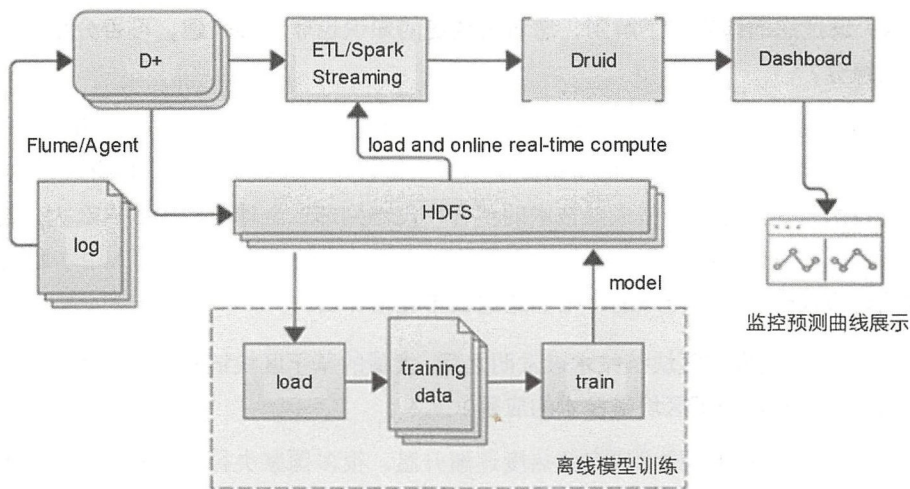
彭冬: Hubble 1.0 (监控平台部分) 底层计算部分的整体架构示意图如图 A-2 所示。



图A-2 Hubble 1.0（监控平台部分）底层计算部分的整体架构示意图

CSDN：不管是大数据平台（D+）还是Hubble，都有哪些机器学习的应用？

彭冬：说到大数据，不可避免地就要提到数据挖掘和用户画像。D+以其强大的数据平台工程体系，在存储和计算上可以支撑广告的用户画像和数据挖掘的能力，当然也依赖公司构建的集群基础设施如 Spark、TensorFlow 等。在 AIOps 上，我们通过 LSTM（长短期记忆网络）尝试使用动态阈值来进行监控报警。由于 LSTM 能很好地抓住时间序列上下文可能存在的联系特性，因此在模型训练方面，我们选择了 LSTM 模型。在 Python 中有不少包可以被直接调用来构建 LSTM 模型，比如 Keras、TensorFlow、Theano 等，我们选用 Keras 作为模型定义与算法实现的机器学习框架，将 Keras 集成到 TensorFlow 中，让 Keras 变成 TensorFlow 的默认 API。让 Keras 运行在 TensorFlow 框架上，可以帮助我们快速搭建和实现一个神经网络模型。同时使用均方误差（Mean Squared Error）作为误差的计算方式，并采用 RMSprop 算法作为权重参数的迭代更新方案。基于机器学习的趋势预测架构示意图如图 A-3 所示。



图A-3 基于机器学习的趋势预测架构示意图

CSDN: 您做了非常多的架构工作, 可否谈一下您对架构的理解?

彭冬: 我觉得架构要做到三点。一是架构要适应业务的发展, 我们在 D+ 和 Hubble 上进行了多次迭代和架构优化, 这是随着业务的不断发展演进而进行的, 鸟枪换炮是一个过程; 二是架构要有一定程度的前瞻性, 要借鉴业界的做法, 吸纳别人的长处, 当然这里面也有不少系统工程问题, 比如如何考虑系统的可扩展性和兼容性; 三是架构存在一定的方法论, 架构应该是方法论与实践论的结合。纸上得来终觉浅, 绝知此事要躬行。

CSDN: 一个架构师需要具备哪些技能或素养?

彭冬: 我经常跟团队成员分享一个架构师的成长阶段。第一个阶段应该是习惯养成, 注重培养良好的编码和设计习惯, 注重代码量和代码质量, 注重抽象; 第二个阶段应该是培养模块和系统设计能力, 在关键系统设计上总结方法、积累经验, 懂得资源 (网络 I/O、磁盘 I/O、内存、CPU 等) 如何最优化利用, 以及系统如何分层, 掌握可用性、高并发性、高性能的一些方法论和设计技巧; 第三个阶段是培养业务思维, 能站在业务需求的角度思考系统架构的合理性和架构演进方向, 能发现影响业务发展的系统瓶颈; 第四个阶段是架构师应具备产品思维。

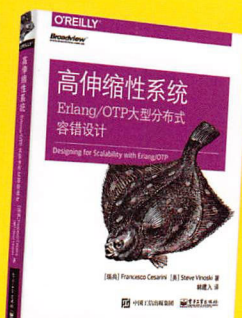
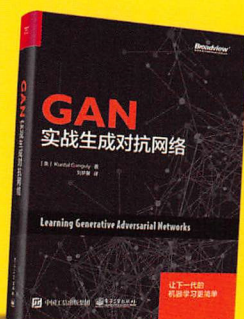
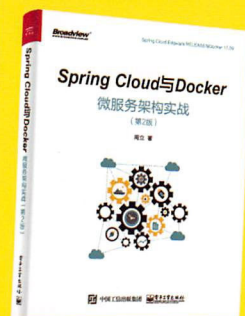
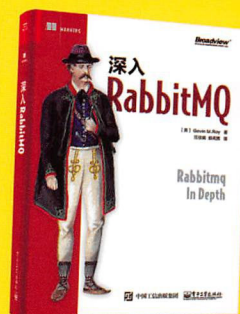
这四个阶段也决定了架构师的四个层次, 大部分架构师都停留在第二个阶段和第三个阶段, 第四个阶段的架构师非常稀缺, 产品思维要求架构师能够具备极其强大的系统抽象能力, 将系统转化为产品, 产品能非常方便地被使用。如果说前面的阶段是把系统做得足够复杂, 那么第四个阶段就是把系统做得足够简单。所谓架构师的终极目标是恶心自己 (设计了非常强大的系统), 成全别人 (非常易于用户使用), 就是这个道理。

CSDN: 通过您的博客我了解到，您对区块链的知识也非常感兴趣，可否分享一下您和区块链技术的缘分？

彭冬: 其实我很早就接触过比特币，2013 年挖过比特币，差点还做了比特币交易平台方向的创业。2016 年我开始深入研究比特币底层的区块链技术，发现原来技术也有玩得这么溜的，其实区块链技术只是进行了一些成熟技术的整合，没想到却在全球范围影响这么大。这里面有一件比较有意思的事情，我经常拿来鼓励我们的工程师，就是比特币的创始人中本聪大概是 60 岁时完成比特币代码，风靡全球的，咱们还这么年轻怕什么。

CSDN: 比特币的底层区块链技术被发明之后，大量的基于区块链技术的应用被开发出来，可否从您的视角来分析一下区块链技术的应用？

彭冬: 这两年对区块链技术研究的热度逐渐升温，很多国家央行等政府部门都开始重视起来，包括微软、IBM、谷歌、百度、腾讯、微博在内的各大企业也纷纷入场，这也印证了区块链技术的热度。因为具有去中心化、安全性、时间戳等机制和特点，能够保证系统容错和公平性，保证信息不可篡改，区块链技术被应用在数字货币、版权、预测、社交、授权等方向。目前微博广告团队正在研究如何建立可信网络联盟，以便在各个广告主、第三方监测机构和微博广告平台之间建立客观公正的监测机制；同时我们也在研究在大数据场景下，如何使用区块链技术建立公平的自动化的数据市场交易平台等。



本书是对AIOps的深度细化和技术补充，相关实践可落地，很有说服力。

—— 萧田国
高效运维社区发起人
AIOps标准及白皮书发起人

AIOps是运维领域的极大热点，本书对底层技术进行详细分析，并结合微博场景提供大量实战案例，非常有参考价值。

—— 裴丹
清华大学计算机系长聘副教授
青年千人 美国AT&T研究院前主任研究员
智能运维算法专家

本书作者对智能运维技术体系进行全面梳理，完整呈现从思路到工具再到实践的全过程。

—— 王鹏云
多盟联合创始人
蓝色光标技术创新孵化中心总经理

本书从大数据技术讲到AI运维，详细介绍实施智能运维依赖的基础设施和架构技术，兼具参考性与实操性。

—— 梁定安
腾讯运维技术总监 专家工程师

本书可作为运维工程师提升运维水平的重要参考，也可作为通过运维+AI向自动化智能运维发展的依据。

—— 钟华
美团打车技术研发部负责人

本书介绍异常检测、根因分析、时序预测等智能运维实践经验，并梳理了其两大基石：大数据和机器学习。

—— 饶琛琳
日志易产品总监
前新浪微博系统架构师

从运维平台大数据处理到架构设计原理，再到AIOps相关模型和算法，将智能运维工程架构与算法实践完美结合。

—— 陈晓峰
火币集团副总裁

本书系统介绍大数据采集、存储、处理、计算及策略应用各环节，并以微博监控为案例展示了监控平台建设实践。

—— 陆沛
滴滴打车SRE团队负责人 技术专家

上架建议：自动化运维/人工智能



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨
责任编辑：葛娜
封面设计：吴海燕

ISBN 978-7-121-34663-7



定价：79.00元